USENIX

FREENIX Track: 2005 USENIX Annual Technical Conference

Anaheim, CA, USA, April 10–15, 2005

conference

proceedings

FREENIX/ Open Source Track

2005 USENIX Annual Technical Conference

Anaheim, CA, USA
April 10–15, 2005

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $30 for members and $40 for nonmembers.
Outside the U.S.A. and Canada, please add
$18 per copy for postage (via air printed matter).

ISBN 1-931971-28-5

USENIX Association

Proceedings of the FREENIX/Open Source Track

2005 USENIX Annual Technical Conference

April 10–15, 2005
Anaheim, CA, USA

2005 USENIX Annual Technical Conference
FREENIX Track
April 10–15, 2005
Anaheim, CA, USA

Index of Authors

Message from the Program Chair

Free software is a topic of special interest to me. It enables people from all over the world to work together and create infrastructure that increases our productivity and improves our daily lives. The entrance barrier for open source is extremely low. We do not need expensive lab equipment—a modest computer is completely sufficient. FREENIX strives to provide a medium for the open source community to come together, communicate with one another, and grow stronger.

At the same time that open source is becoming more accepted by society, the dangers to its survival increase. Litigation in the USA and pending software patents in Europe will restrict the freedom of our community if they come to pass. Fortunately, public awareness of these challenges is increasing, and it is my hope that open source will continue to flourish for many years to come.

In the 8th year of the FREENIX track, we are proud to present you with a very strong program. We received 56 submissions, ranging from four-page extended abstracts to full-length papers. Each submission received at least five reviews from the Program Committee and external reviewers. At the end of the Program Committee meeting in December 2004, we were able to accept 18 exciting papers covering a broad range of topics. Unfortunately, we had to turn down a number of strong submissions, but we would like to encourage the authors to resubmit their papers to next year's FREENIX track.

As the Program Chair, I would like to thank all the authors for their hard work. Furthermore, many thanks are due to the Program Committee and reviewers who spent a large amount of time carefully reading and evaluating all submissions. The proceedings that you have in your hand today are the result of their thoughtful feedback and help in shepherding papers along to their final polished stage. Of course, without the consistent support and help of the USENIX staff none of this would have been possible.

I hope to see all of you at the Annual Technical Conference in Anaheim and am looking forward to the opportunity to discuss recent developments and exciting work in open source with you.

Niels Provos
FREENIX Program Chair

Build Buddy For Fun And Profit

Dan Mills
Novell, Inc.

Abstract

We present a build and packaging system called Build Buddy. The system is comprised of a set of tools for building and maintaining software packages on multiple operating systems and architectures.

1 Introduction

Release engineering can be a complex and time-consuming task. Distribution vendors often employ multiple people to work full-time on building and packaging software. Ximian (Now a part of Novell) was faced with this problem as well, when releasing Ximian GNOME, a customized version of GNOME and other add-on applications. Ximian wanted to reach the widest possible audience, and that meant releasing on multiple platforms.

As any release engineer is aware of, releasing software on multiple platforms requires a good deal of effort. In addition to the complexities added by having different possibly incompatible versions of build dependencies (required libraries, tools, or other files), different distributions sometimes use different packaging systems, and even when they dont, they often have different packaging policies, which, if they are to be followed, add a new layer of requirements to be met.

Ximian took a stab at this problem, and created a collection of tools called Build Buddy to automate the process of building and packaging software. These tools allow release engineers to produce packages for multiple operating systems, verify package corectness, and submit packages to an external repository (such as a Red Carpet server). Building can be done at the push of a button, on a schedule, or continuously. Release engineers can also create customized build reports to be as plug-ins, or check job status through a variety of ways (from a Web UI to log files on disk).

2 Design And Implementation

The basic watchwords of build-buddy are “automation” and “reproducibility”. Build Buddy’s goal is to make it possible not only to build complex software packages (and test them) with a few simple commands, but to make it possible for developers creating the package to use the same process as the automatic world-building engines. Moreover, Build Buddy is designed to abstract the details of the underlying packaging system by encapsulating the build commands and packaging metadata in an XML file, which gets translated to the local packaging system of the current distribution. In this way, Build Buddy produces native packages which are installable by users without requiring any additional package management software. This is an important difference over most other cross-platform build environments: Build Buddy is designed to produce packages that tightly integrate into the systems they will ultimately be deployed to.

On a higher level, Build Buddy is responsible for setting up and enforcing protected build areas called jails, so a single machine can reliably build multiple copies or versions of products without stepping on anyone’s toes by clobbering libraries or falling victim to version skew of installed dependencies.

On an even higher level, Build Buddy enables the build and packaging process to take place on a remote machine by providing an XML-RPC interface to a build daemon that waits for build requests, as well as a “master” scheduling daemon which is used to keep track of multiple build nodes and relay a build request to an appropriate node. There are web and command-line interfaces for submitting build job requests to the master, as well as the ability to mark a job to be re-run routinely as a snapshot.

2.1 Packaging System Abstraction

Build Buddy can produce native packages for the following packaging systems[1]:

- RPM[2] (Red Hat, SuSE, Mandrake, and others)
- DPKG[3] (Debian)
- SD[4] (HP-UX)

To achieve this, we use an XML description of the packaging metadata that will ultimately be converted to an RPM `spec` file, a DPKG `debian/rules` file, or an SD `psf` file. Although the format of all three vary quite a bit, much of the information is similar. For those cases when the information needs to be different per-platform, Build Buddy allows the release engineer to override chunks of the XML on a per-target basis.

2.1.1 Targetsets

Targetsets are a major feature of the Build Buddy XML package description file. They allow the writer to selectively apply chunks of XML to certain platforms, with a very simple syntax. There must be one targetset, called the "default" targetset, that matches everything:

```
<targetset>
    <filter>
        <i>.*</i>
    </filter>

    ...XML here...

</targetset>
```

After it, any number of additional targetset sections may be defined, matching any combination of targets:

```
<targetset>
    <filter>
        <i>suse-91-i586</i>
        <i>sles-.*-ppc</i>
        <i>fedora</i>
    </filter>

    ...XML here...

</targetset>
```

When the XML file is read, all matching targetsets are merged together, thus allowing the release engineer to place in the default targetset as much information as is common among different targets, and adding, removing, or changing those default on specific sets of operating systems or architectures.

[1]There is also IRIX Inst support, though largely unused. RPM platforms are the only ones supported for `bb_node`.

[2]http://rpm.org/

[3]http://debian.org/

[4]http://software.hp.com/products/SD_AT_HP/

2.1.2 Macros

Macros are a way of avoiding duplication of information. Our XML files make extensive use of these string replacement macros. Commonly, they are used to define parts of paths and other useful variables. For example, the `[[prefix]]` macro can be set-up to expand by default to `/usr` on various Red Hat Linux OSes, but to `/opt/gnome` on SuSE Linux OSes. This way, the command:

```
./configure --prefix=[[prefix]]
```

Will cause the software to be configured differently depending on the target being built on. Moreover, macros can reference other macros (as long as there are no recursive definitions), so it is possible to e.g., define a `[[configure]]` macro, which sets the prefix, sysconfdir, localstatedir, and other miscellaneous settings as desired on each target.

Macros are set in a global configuration file for all of Build Buddy, but they can be overridden on a per-module basis. When this is done, other macros that reference the overridden macro will use the new definition. So for example, if a module defines:

```
<macro id="prefix">/opt/myproduct</macro>
```

The `configure` macro will correctly configure the product to run from `/opt/myproduct`.

2.1.3 Project Organization

Build Buddy was designed to maintain a packages created from 3rd party sources. For this reason, the packaging metadata is kept separate from sources and patches. Each XML build file and its associated sources, patches, and any auxiliary files is called a "module".

In the general case, only a single XML configuration file per module is needed. Occasionally, however, other files must be provided. For this reason, Build Buddy projects are organized in CVS with one directory per module, and all of them are (usually) placed inside one CVS module.

2.2 Sources And Patches

Build Buddy uses a simple source repository, which can be set-up locally or remotely (using ssh) with minimal configuration. Files may be placed in the repository by using the `bb_submit` tool, which returns a repository handle that may be inserted in a `ximian-build.conf` file or used with the `bb_get` tool to retrieve the file. It is not possible, however, to remove a file from the repository. When an updated version of a file with the same name exists, it is simply submitted again, and a new handle is produced–but the old handle is still valid.

The repository can be used to hold patches as well. These are treated equally by the repository, but they are used in a `<patch>` section in the XML packaging metadata, instead of a `<source>` section.

There exist several tools in Build Buddy to manage patches. While Build Buddy's patch management capabilities were not designed to supplant a version control system, it is possible to maintain source changes completely within Build Buddy. The `bb_regenerate` tool can generate a new patch by comparing a modified source tree to a pristine one. It can also regenerate a patch if it already exists. For example, if the sources have the following patches applied:

1. add-french-translation.patch-2
2. fix-autoconf.patch-1
3. fix-bug-51242.patch-1

And the second patch needs to be changed, it is possible to make those changes and ask `bb_regenerate` to create a new version of it, even if it partially overlaps with other patches. The new patch can then be tested and submitted to the repository (which will create the `fix-autoconf.patch-2` handle).

Build Buddy supports other source acquisition mechanisms in addition to the simple repository. HTTP and FTP behave similarly to the standard repository, but URLs are used instead of a repository handle.

It is also possible to use CVS or Subversion to acquire sources. In this case, Build Buddy will checkout the sources and attempt to create a distribution tarball. This command is configurable via the `<dist>` tagin the XML configuration file. If not specified, it will attempt to run `autogen.sh`, and `make dist`. Once a tarball has been created, it is used as the source for the rest of the build, and for the source packages. This practice ensures that a valid source package is created as part of the build process.

2.3 Basic Build Process

When building a single module, the following steps are generally taken (see fig. 1):

- Parse the XML build config file.
- Acquire all sources as defined.
- Produce local build/packaging files (e.g., RPM .spec).

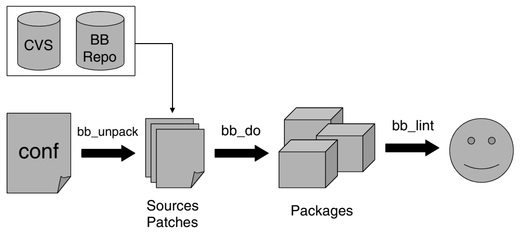


Figure 1: The basic build and packaging process

- Build and install to a temporary location.
- Package the built files.
- Test the resulting packages.

Build Buddy executes as many steps as possible from within the packaging system's environment. For example, on RPM systems, the build commands are executed through RPM. This helps ensure the source package will be valid and usable by third parties. However, when the packaging system does not support a certain step, Build Buddy performs it directly. For example, the SD system does not have a concept of a source package and simply expects binaries as its input files. In this case, Build Buddy emulates the RPM behavior by executing the build commands directly, and creating an additional package with sources.

2.4 Verifying Package Correctness

Even the best of packagers is bound to make a mistake sometime. To catch as many errors as we can before letting a package out the door, Build Buddy has a tool called `bb_lint`, which can be thought of as a unit testing framework, but restricted to packaging policy/metadata. It is possible to easily create new tests to be run either globally or for a particular module, or to disable a check for a particular module. Examples of existing tests include:

- Whether all files the module installs have been packaged.
- Whether there are any files in the package that are not owned by a system user (such as 'root', etc).

- Whether all packages have a "group" defined.

Some lint tests are designed to be run before the package has been built, such as the "group" check. Others, require the packages to be created, such as the test for non-packaged files. Tests can be flagged as errors, or warnings.

2.5 Building Large Products

In addition to the ability to build a single module on many operating systems, Build Buddy can also help manage the complexities of a large product with interdependent modules. It is possible to create another XML file called a `product` file, which describes the build-dependencies of all the modules involved. Then, one can use the `bb_build` tool to perform arbitrary operations (operations can be defined via plug-ins) on any subset of the module graph. For example, given the modules:

- `glib`
- `gtk+` (depends on `glib`)
- `nautilus` (depends on `gtk+`)
- `tcpdump`

It is possible to request a rebuild of `glib` and everything that has a dependency on it (`gtk+` and `nautilus`, but not `tcpdump`).

Due to the flexible operations, however, it is also possible to perform many other tasks that span multiple modules through `bb_build`. Examples include creating a HP-UX PSF file for a Bundle (a collection of products), running only specific portions of the build process (such as `bb_lint` by itself), or custom operations, such as unpacking a module and determining if it contains a certain source file or not.

Custom operations do require some programming and knowledge of certain Build Buddy data structures, but care has been taken to keep them easy to create and deploy. It is also possible to create a 'task', a sequence of operations that can then be conveniently invoked by a single name. For example, the default operation `bb_build` runs is 'build', which is a task that unpacks, builds, runs `bb_lint`, and cleans up the built tree. Thus, to execute the default sequence and add something at the end, the user can specify "build,oper" as the operation.

Upon encountering an error, `bb_build` can stop all operations on the current module as well as modules that depend on it. This is the defautl behavior. The user can also specify more or less lenient behavior, that is, to attempt to build dependent modules, or to halt immediately on error.

`bb_build` allows all output (including the output of the build process itself) to be sent to a file. This can be combined with home-grown scripts to integrate Build Buddy into a build process without using the XML-RPC Build Buddy daemons.

2.6 Jails

Jails are extensively used in Build Buddy. A jail image is essentially a tarball of an entire distribution, plus some metadata. When a jail image is unpacked, the chroot system call is used to enter the jail, so the build node is protected to some degree from the build, and vice-versa. However, as anyone familiar with chroot is aware, chroot does not provide a great deal of insulation–a number of things are shared, such as the process table, etc. Still, as a poor man's VM, it is excellent.

Build Buddy keeps arbitrary XML metadata associated with the jail image and with unpacked jails. Metadata currently used include hints to the node to control automatic deletion of old jails, information about the jail's owner, information about mount points that Build Buddy will automatically mount (such as /proc), etc. Since the metadata is arbitrary and easily parseable, external tools can tag jails, for example, for archival.

When the remote interface (Web UI / XML-RPC) is being used, it is possible to specify via XPath queries certain values to be met in the to-be-used jail. This makes it possible, for example, to deploy a specialized jail to be used for certain build jobs.

2.7 XML-RPC Interface

The components described so far operate on one or more modules, on a single machine or jail. Build Buddy also includes a set of networking components designed to automate the build process and manage a "build farm". In this set-up there is one "master" and many build "nodes". Each node registers itself with the master, which keeps track of them, and serves as a scheduler to distribute build requests, as well as a centralized point to collect logging information.

The current XML-RPC interface does not allow for full `bb_build` usage remotely. Instead, build requests specify each module to be built, in order. There are, however, plans to improve on this point by extending the XML-RPC interface.

The structure of the Build Buddy networked daemons is as follows (see fig. 2):

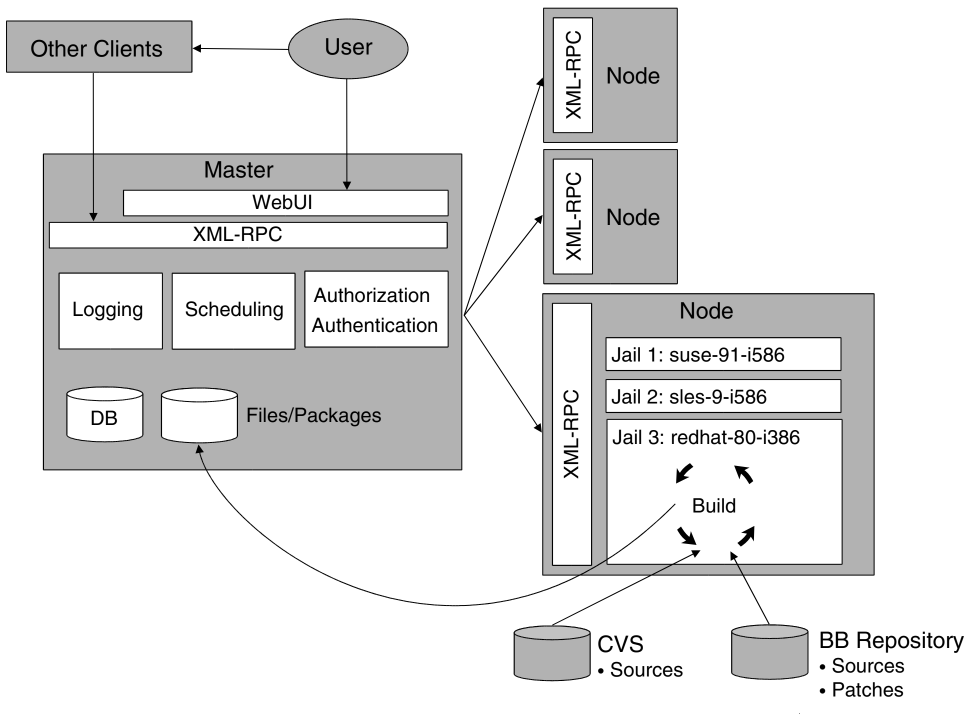


Figure 2: The networked build farm

The master is the main user-visible machine. Its purpose is to monitor the nodes, collect information about builds, and provide a single point of contact for the system. As such, it runs various daemons, including a web server, a database, and several Build Buddy services.

These services are:

- The logger is responsible for collecting all logging output from the jobs and storing it on-disk.
- The master is responsible for the scheduling of jobs, and for presenting a unified XML-RPC interface for all the available nodes.
- The authserver is responsible for user authentication, and also for authorizing or denying any requests from users.
- The snapshotter is an optional service to run saved jobs marked by users to be executed automatically on a regular basis.

The node is an individual machine which can build software upon request. Jails only need to run one background process, the `bb_node` daemon itself.

The actual build, which runs inside a jail, uses CVS to acquire the `ximian-build.conf` files. The sources to be built are defined in those files, and can point to external CVS, HTTP, FTP servers, or a Build Buddy file repository.

The build can also specify build dependencies, which are installed using the `Red Carpet` utility.

2.8 Remote Build Process

When a build request is submitted via the XML-RPC interface, the following steps take place:

1. The build master schedules the job to a node, taking into account the architecture requested, number of running jobs on each node, free HD space, etc.

2. The node searches for an available build jail for the requested os-version-arch (which Build Buddy calls a “target”). If none are available, a new one is unpacked.

3. Red Carpet is set-up inside the jail, to provide a method of installing build dependencies.

4. The XML build configuration files are checked out of CVS.

5. For each module to be built, these steps are performed:
   (a) Build dependencies are installed if needed.
   (b) The sources of the module are obtained from the tarball repository, or from CVS/Subversion.
   (c) `bb_lint` is run to check the XML metadata.
   (d) A `spec` file is generated for RPM. [5]
   (e) The software is built and packaged as per the `spec` file.
   (f) The resulting packages are synced over to the master (to avoid having to set-up a shared filesystem between the nodes and the master, rsync is used).
   (g) The resulting packages are tested for packaging errors with a second `bb_lint` run.
   (h) The resulting packages are installed inside the jail (which is necessary, since they can provide a build dependency for a later module).
6. Finally, if all went well, the packages are optionally submitted to a Red Carpet server, so that users can get at them, and so they can be used as build dependencies for other build jobs.

2.8.1 Logging

Throughout the entire process above, the status of the job, including up-to-the-minute process output logs can be seen via the Build Buddy Web UI. Nodes collect process output, and relay it to the master via a specialized XML-RPC daemon, which writes it to disk[6]. Reporting modules called "logstyles" are also available for snapshot jobs. Logstyles are event-based, and are highly customizable. Currently available logstyles include support for email reports (both text and HTML), and RSS.

2.8.2 Job Submission

There are three[7] interfaces to submit a build job to the master. One is a command-line interface called `bb_client`. The second is also a command-line utility, but it is designed to run on the master, for snapshot (recurrent) jobs, that is the `bb_snapshot` tool. The last method is the web interface. The web interface allows users to specify job information and save it to a database on the master. They can then submit the job, or they can mark it for snapshotting (which gets picked up by `bb_snapshot` when it runs).

[5]The node does not currently support Debian, HP-UX, or Solaris, even though the lower-level tools do.

[6]Previous Build Buddy versions used NFS to do this work, but it made the system harder to deploy, and was not more reliable than the current method.

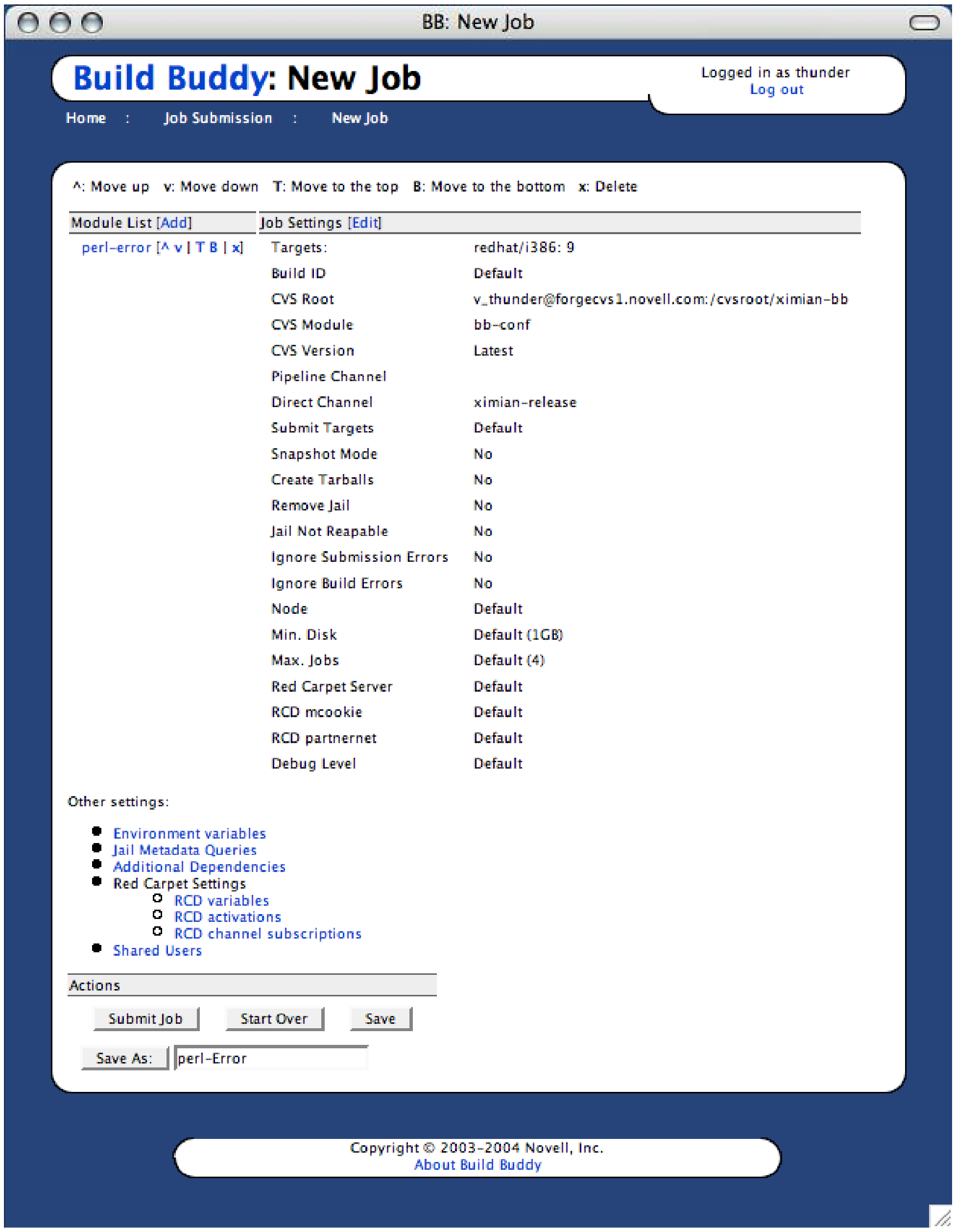
[7]Actually, there are two more. One is an Emacs Lisp program to submit the XML config currently being edited for building, but it has not kept up with the latest XML-RPC interface changes. The other is a plugin for the Eclipse IDE to integrate with a Build Buddy installation at Novell Forge, which due to its Novell Forge-specific nature is not bundled with Build Buddy

3 Example Usage

The following is a working example of an XML build configuration file to build and package the 'Error' Perl module:

```
<?xml version="1.0" ?>
<!DOCTYPE module SYSTEM "helix-build.dtd">

<module>
    <targetset>
        <filter>
            <i>.*</i>
        </filter>

        <rcsid>$ Id: $</rcsid>
        <name>Error</name>

        <version>0.15</version>
        <rev>1</rev>
        <serial>1</serial>

        <psdata id="copyright">Artistic</psdata>
        <psdata id="url">http://www.cpan.org/</psdata>

        <source>
            <i>Error-0.15.tar.gz-1</i>
        </source>

        <build id="default">
            <prepare>[[perlprepare]]</prepare>
            <compile>[[perlmake]]</compile>
            <install>[[perlinstall]]</install>

            <package id="default">
                <name>perl-Error</name>
                <psdata id="group">Development/Perl</psdata>

                <files>
                    <i>[[perlmoddir]]</i>
                    <i>[[usrmandir]]/man*/*</i>
                </files>
                <docs>
                    <i>README</i>
                </docs>

                <description>
<h>Error extension for Perl5</h>
<p>This package provides a perl module.</p>
                </description>
            </package>
        </build>
    </targetset>
</module>
```

The above configuration file is placed in CVS, as described in the "Project Organization" section. To use the scripts directly, the conf is checked out on the build machine, and the various scripts (such as `bb_build`, `bb_do`, etc) are executed. The release en-

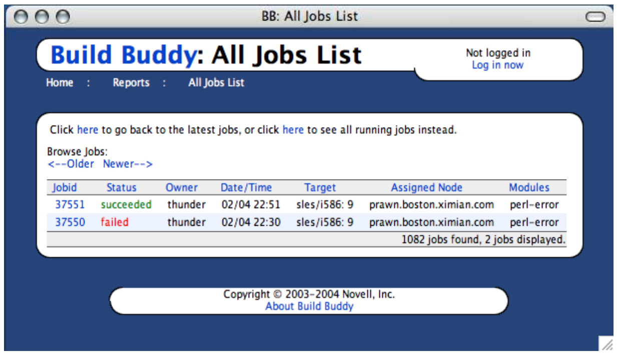

BB: New Job
Build Buddy: New Job
Logged in as thunder
Log out
Home : Job Submission : New Job
^: Move up v: Move down T: Move to the top B: Move to the bottom x: Delete

Module List [Add]	Job Settings [Edit]	
perl-error [^ v | T B | x]	Targets:	redhat/i386: 9
	Build ID	Default
	CVS Root	v_thunder@forgecvs1.novell.com:/cvsroot/ximian-bb
	CVS Module	bb-conf
	CVS Version	Latest
	Pipeline Channel	
	Direct Channel	ximian-release
	Submit Targets	Default
	Snapshot Mode	No
	Create Tarballs	No
	Remove Jail	No
	Jail Not Reapable	No
	Ignore Submission Errors	No
	Ignore Build Errors	No
	Node	Default
	Min. Disk	Default (1GB)
	Max. Jobs	Default (4)
	Red Carpet Server	Default
	RCD mcookie	Default
	RCD partnernet	Default
	Debug Level	Default

Other settings:

- Environment variables
- Jail Metadata Queries
- Additional Dependencies
- Red Carpet Settings
  - RCD variables
  - RCD activations
  - RCD channel subscriptions
- Shared Users

Figure 3: Job Submission



Build Buddy: All Jobs List

Click here to go back to the latest jobs, or click here to see all running jobs instead.

Browse Jobs:
<--Older Newer-->

Jobid	Status	Owner	Date/Time	Target	Assigned Node	Modules
37551	succeeded	thunder	02/04 22:51	sles/i586: 9	prawn.boston.ximian.com	perl-error
37550	failed	thunder	02/04 22:30	sles/i586: 9	prawn.boston.ximian.com	perl-error

1082 jobs found, 2 jobs displayed.

Copyright © 2003-2004 Novell, Inc.
About Build Buddy

Figure 4: Recent Jobs (shortened list)

gineer can then use the resulting packages however is desired.

When using the WebUI, a new job is created for perl-Error (see fig. 3). The job includes information on where to check out the configuration file from, and which operating systems/platforms to build it on, as well as a Red Carpet server and channel to submit it to.

Since the configuration file specifies a tarball in the `<source>` tag (to be retrieved out of the Build Buddy file repository), this module will likely not be run as a snapshot. We can, however, update the configuration file stored in CVS at any time, increase the version as appropriate, and simply re-submit the job to build the new packages.

Once a job is submitted, it will appear under the "recent jobs" list (see fig. 4). The user can click on the job id to bring up job details (see fig. 5). From this page, it is possible to view individial log files, as well as download any generated packages, by clicking on them.

4 Future Plans

Although a great deal of effort has been spent on Build Buddy, there is still much that can be done. Aside from minor improvements in all areas, the biggest projects currently under consideration are a flexible way to keep track of a project's modules in the WebUI, and a redesign of the packaging system abstraction XML to allow for new platform support andd additional features.

4.1 WebUI Improvements

The WebUI is currently limited to a rigid definition of a build job, and has no definition of a project from which jobs can be derived. In reality, projects with multiple components will want to keep track of all the components without associating them to a "saved job", and be able to produce jobs based on the project data with less set-up than is currently needed.

To accomplish this, a logical container of modules can be created (the project), and the UI can be modified to allow build queries which allow the release engineer to make use of the bb_build command remotely.

4.2 Expanded Platform Support

Build Buddy's most unique feature is its ability to produce native packages on a wide range of platforms. Currently, this is restricted to UNIX-like operating systems, but there is no reason why this should continue to be the case. A research prototype exists that is able to build MSI packages on Microsoft Windows. This presents some interesting challenges, due to the differences between RPM, Dpkg, or SD, and Windows' MSI, but they still have enough in common that it is possible to use a single configuration file for all.

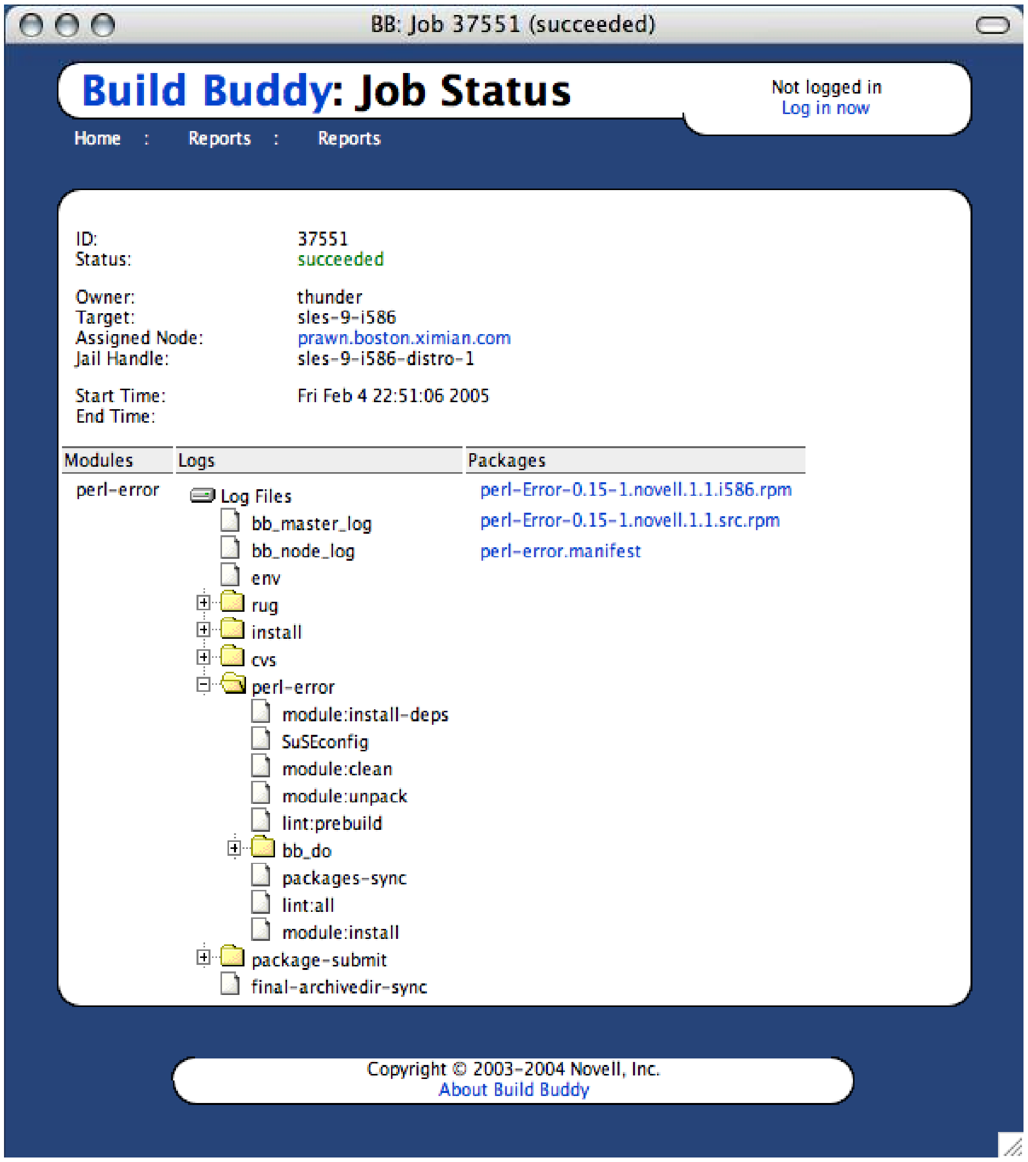
The Windows prototype is written in C#, and a new XML abstraction format was designed, both to add data only useful to MSI packaging, as well as to fix other long-standing problems or annoyances with the current XML format being used in production systems.

The MSI format is particularly different in its rich set of metadata for arbitrary files or folders. RPM and Dpkg do not generally require much additional metadata for files and folders. Flags for configuration files, documentation, permissions, or the like are sufficient in most cases, and several of these are mutually exclusive, making representation simpler.

The XML format implemented in the prototype has been designed to allow for arbitrary metadata for any resource (file, directory, or other) to be added at any later date. This is done by substituting the current file lists, which are simple strings, with a richer XML representation. At the same time, since string substitutions (Build Buddy macros) will not be useful for file lists, methods were added to produce the same results, while avoiding the need to be verbose.

4.3 Build Node Imaging

There are certain types of builds or platforms for which it would be best to completely convert a node to a different distribution, rather than use a Build Buddy jail. Some distributions make use of specific kernel features, for example, which prevent them from working properly in a chrooted environment. A solution to this problem is to use a technique generally



ID:	37551
Status:	succeeded
Owner:	thunder
Target:	sles-9-i586
Assigned Node:	prawn.boston.ximian.com
Jail Handle:	sles-9-i586-distro-1
Start Time:	Fri Feb 4 22:51:06 2005
End Time:	

Modules	Logs	Packages
perl-error	Log Files	perl-Error-0.15-1.novell.1.1.i586.rpm
	bb_master_log	perl-Error-0.15-1.novell.1.1.src.rpm
	bb_node_log	perl-error.manifest

```
Log Files
    bb_master_log
    bb_node_log
    env
  + rug
  + install
  + cvs
  - perl-error
        module:install-deps
        SuSEconfig
        module:clean
        module:unpack
        lint:prebuild
      + bb_do
        packages-sync
        lint:all
        module:install
  + package-submit
    final-archivedir-sync
```

Figure 5: Job Details

known as “imaging”. This technique uses disk images and network booting to achieve the equivalent of a full re-install of the operating system. Build Buddy could leverage this technology to augment or replace the current Jail system.

5 Related Work

Build Buddy is not the only project with the ability to build software and package it. One such tool is Maven (http://maven.apache.org/). Maven is a Java-based tool to help manage and build projects. It stores information about your project, from the names of developers to lists of dependencies, in an XML format, which it can then use in a variety of ways.

Another tool is SCons (http://www.scons.org/). SCons is a make, autoconf, and automake replacement, written in python. As with the tools it replaces, it is designed to precisely track the build process in detail, as well as build dependencies.

The biggest difference between Build Buddy and the above projects is that they concentrate on the build step, while Build Buddy’s emphasis is actually on the packaging. It is possible to write scripts using any of the above tools to package software, but it would be a custom, per-project solution.

All of the above tools provide much greater detail and control over the specifics of the build process– that is, they solve problems such as: Which files need to be built before others? Exactly what commands need to be run to compile each file? Build Buddy, on the other hand, limits itself to invoking other tools to perform this work. In other words, although Build Buddy orchestrates the build process at a macro level, it invokes tools such as make, automake, or maven to achieve the specific build steps.

In this sense, Build Buddy is able to leverage the advantages of all of the above tools. Software authors are able to use Maven, SCons, make, etc. and simply invoke those tools from Build Buddy as needed. Build Buddy then focuses on the cross-distribution software packaging. Indeed, many software modules at Novell which are built using Build Buddy use make, autoconf, automake, or maven.

The disadvantage of this approach is that software authors who wish to use Build Buddy must use a combination of tools, rather than a single one, to perform the full build and packaging set of tasks. On the other hand, release engineers who integrate software from disparate places will find the Build Buddy’s ability to adapt to any build tool highly desirable.

6 Summary

After more than four years of development, Build Buddy useful for a wide range of packaging tasks. By using all of its components, an experienced packager can develop and maintain large collections of packages for a number of distributions at the same time. At the same time, relatively novice packagers or developers, can use Build Buddy to make one-off packages, or maintain small sets of software with a minimum amount of training.

Build Buddy is licensed under the GNU GPL. Source code and additional documentation available at http://build-buddy.org/. We hope that you find it useful.

Acknowledgments

Many thanks to the original Build Buddy author, Mike Whitson, as well as Peter Teichman, Dave Camp, and other contributors. Thanks also to all Build Buddy users who have supported the project and helped shape it into what it is today. Last but not least, thanks to Nat Friedman and Miguel de Icaza, and other leaders at Ximian who had the foresight to invest in the project.

References

[BNS+00] Edward C. Bailey, Paul Nasrat, Matthias Saou, Ville Skyttä. Maximum RPM, 2000.

[JS98] Ian Jackson, Christian Schwarz. Debian Policy Manual, 1998.

[QR+01] Daniel Quinlan, Paul Russell, Filesystem Hierarchy Standard Group. Filesystem Hierarchy Standard 2.2, 2001.

[HP01] Hewlett-Packard Company. Software Distributor Administration Guide for HP-UX 11i, June 2001.

[ASF] Apache Software Foundation. Maven Project, http://maven.apache.org/.

[SF] The SCons Foundation. SCons Project, http://www.scons.org/.

[FSF] The Free Software Foundation. GNU Autoconf and GNU Automake, http://www.gnu.org/.

Scmbug: Policy-based Integration of Software Configuration Management with Bug-tracking

Kristis Makris
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ, 85287
kristis.makris@asu.edu

Kyung Dong Ryu
IBM T.J. Watson Research Center
1101 Kitchawan Rd
Yorktown Heights, NY 10598
kryu@us.ibm.com

Abstract

Software configuration management(SCM) and bug-tracking are key components of a successful software engineering project. Existing systems integrating the two have failed to meet the needs of the ASU scalable computing lab, powered by open-source software. An improved solution to the integration problem, designed to accomodate both free and commercial systems alike, is presented.

Scmbug offers a policy-based mechanism of capturing and handling integration of SCM events, such as committing software changesets and labeling software releases, with a bug-tracking system. Synchronous verification checks and the flexibilty to match multiple development models separate this approach from related work. We address design limitations of existing integration efforts, suggest improvements in SCM and bug-tracking systems required to achieve a scalable solution, and document our early integration experiences.

1 Introduction

SCM is key[17] in maintaining quality in software engineering. SCM systems, or even simple version control systems, guarantee that a record of all changes and enhancements to software is maintained. When bugs creep in software, a trace of how changes occurred is available, simplifying the process of identifying regression points and correcting defects.

Paired with SCM, bug-tracking is another key in engineering quality software. Defect-tracking systems guarantee that nothing gets swept under the carpet; they provide a method of creating, storing, arranging and processing defect reports and enhancement requests. Those who do not use a bug-tracking system tend to rely on shared lists, email, spreadsheets and/or Post-It notes to monitor the status of defects. This procedure is usually error-prone and tends to cause those bugs judged least significant by developers to be dropped or ignored[10].

By examining a log of software changes from an SCM tool, it is uncertain *why* the changes occurred. By examining a defect report, it is uncertain *what* changed in software in response to the defect. Integration of SCM with bug-tracking ties the reason *why* a feature/defect was developed/fixed with *what* software changes occurred in the SCM system to accomplish this. Marrying the SCM and bug-tracking systems improves the traceability of software changesets, quality of documentation in defect reports, and quality of release documents.

SCM[11, 9, 18, 6, 5, 7, 3] and defect-tracking[2, 4, 1] systems are widely popular and in deployment. Commercial systems implement a mostly inflexible integration of the two, and free systems lack a variety of important verification guarantees. None of the existing systems met the integration demands of the ASU scalable computing lab. At a minimum, an integration system should provide:

- Integration of *common denominator* SCM events, such as committing changesets and labeling releases. When a changeset is committed, the accompanying log message should be inserted in the bug-tracker. The list of affected files, and the older/newer version numbers of each file, should be reflected in the bug report. When a release is labeled, the release version should be inserted in the bug-tracker.
- Synchronous verification checks of SCM

actions against the bug-tracker. If a developer attempts to commit a changeset against an invalid product or bug id, the commit activity should fail, and the developer should be informed immediately.

- Policy-based configuration. The integration system should be able to match the development model followed by an organization by tuning run-time parameters.
- Secure deployment of the integration over the public Internet.
- An interface to integrate any SCM system with any bug-tracking system, overcoming limitations of the involved systems where possible.
- A mechanism to produce a Version Description Document (VDD) detailing the defects corrected, and the changesets involved in fixing a defect, between two releases of a software.

Scmbug is designed to meet these requirements. It is implemented in Perl, an excellent, cross-platform, glue programming language, for UNIX-like systems, such as Linux, AIX, and Solaris. It currently supports integration of CVS and Subversion with Bugzilla.

The rest of this paper is structured as follows. Section 2 outlines the limitations of existing systems that prompted our solution. The system design is presented in Section 3, and a short example in Section 4. The system's components are analyzed in Sections 5, 6, and 7. Section 8 brings to light the improved quality of release documentation enabled by this system. Early experiences using Scmbug are documented in Section 9, and insight to future work is given in Section 10. Finally, Section 11 concludes this paper.

2 Related Work

In the realm of commercial products, Perforce[7] and ClearCase/ClearQuest[3] only offer integration of their own SCM and bug-tracking systems, in a proprietary way. They do not readily support integration with free SCM systems, such as CVS[11], Subversion[9], and Arch[18] or free bug-tracking systems such as Bugzilla[2] MantisBT[4], and AntHill[1]. Work enabling integration of Perforce with Bugzilla 2.0-16 is available[8], but the integration API lacks an abstract bug-tracker interface, and requires modifications released by the Perforce integration team every time the Bugzilla database schema changes. Additionally, integration with other bug-tracking systems requires implementing a communication interface with the target bug-tracker. Finally, the integration is unsuitable for deployment across the public Internet, for reasons explained in Section 5.1.

In the free software arena, the most frequently attempted integration involves CVS[11], the dominant open-source network transparent version control system, with Bugzilla, a free defect tracking software that tracks millions of bugs and issues for hundreds of organizations around the world. Steve McIntyre's website[21] documents an approach using trigger scripts to email CVS's actions to an account (Bugzilla Email Gateway) configured to parse the email and process it accordingly. This approach is not synchronous. If a user accidentally commits against the wrong bug number, or a bug against which he is not the owner, the SCM system will proceed with the commit action regardless. The user will not be given the option to correct her actions. Additionally, if the email gateway is not active, the developer will not be immediately aware that integration failed.

The somewhat dated CVSZilla[15] project also integrates SCM events produced by CVS with Bugzilla 2.10. It does not support integration of events produced by any SCM system in a generic way. Moreover, it modifies the Bugzilla schema, and as a result does not work with current versions of Bugzilla, such as 2.18. Finally, this integration is unsuitable for deployment across the public Internet.

John C. Quillan was first to conceive and implement synchronous verification checks and a VDD generator, in work that was never publicly released. He integrated CVS with Bugzilla, but his design did not support any SCM system with any bug-tracking system, or deployment over the public Internet. Scmbug incorporates these features, and proposes a more flexible design that can accommodate both free and commercial SCM and bug-tracking systems.

3 System Architecture

Scmbug is a client/server system. As shown in Figure 1, it consists of a set of SCM system hooks that capture standard SCM events and a generic glue mechanism of handling these events on the machine hosting an SCM repository. These events are translated into integration requests and transported to a server daemon. The daemon em-

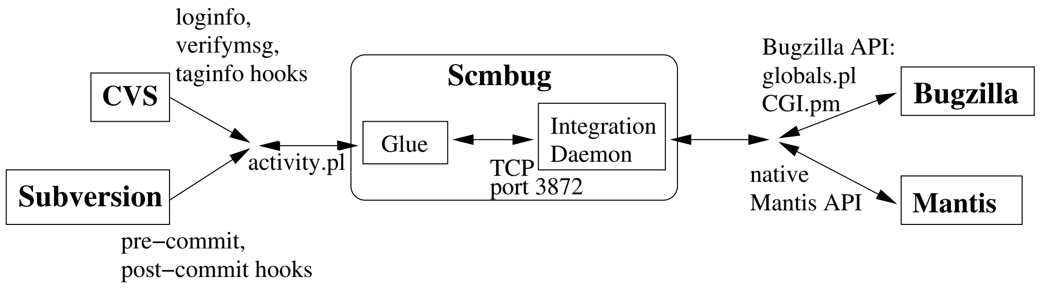


Figure 1: Scmbug system architecture diagram

ploys a generic mechanism of handling these requests and contains functionality that can process these requests per bug-tracking system.

The integration glue follows a common interface of handling SCM events, and groups the logic driving the behavior of the integration. SCM-specific implementation idiosynchracies, such as parameter decoding from SCM hooks, are isolated in separate modules. The abstraction of an integration glue is what permits any SCM system to be integrated using Scmbug. The integration functionality is always executed on the machine hosting the SCM repository.

Respectively, the integration daemon implements a common interface of accepting and processing integration requests. Bugtracker-specific functionality, such as reading metadata and updating bug comments, is implemented in separate modules. The abstraction of an integration daemon is what permits any bug-tracking system to be integrated using Scmbug. The SCM repository and the integration daemon can be hosted on separate machines.

4 Integration example

Lets examine an actual integration sequence using CVS and Bugzilla. Σ-Watch is a performance monitoring tool created by the ASU scalable computing lab. Monitoring the performance of a Zaurus SL-6000 PDA using this tool fails, and this defect is recorded in bug 417. After some initial investigation, the failure point is identified and documented in comment 6. A developer implements a fix, and attempts to commit his changeset using the command `'cvs commit'`. This command brings up a predefined log message template, in which the developer explains the source code change and enters the matching bug id (417) the changeset applies to, as shown in Figure 2.

Figure 3 presents the log comment stored in CVS, produced in a more readable Changelog format using the `cvs2cl` tool, in response to this changeset. This entry clearly details *what* changed: a global variable is introduced to hold the name of the network device. However, it does not explain *why* this change was required. *Why* must a global variable be used instead of a hardcoded value? To identify the reason *why* these files were modified one must consult the bug-tracking system. This log entry refers the reader to bug id 417.

Figure 4 captures the comment entered in Bugzilla in response to the changeset. The original SCM log message is included in the bug comments. The integration also generates a list of affected files, matched with the older/newer version numbers of each file, and appends it to the log message to show *what* changed in response to the bug fix. To identify *why* this changeset occured, one only needs to scroll in past history in Bugzilla to comment 6, with minimal effort, which reveals the root of the problem, as shown in Figure 5: PDAs use wireless, rather than ethernet, for network connectivity. The traceability of software changesets back to the root of problems is dramatically improved.

Note that a complete fix to the bug does not end with a single changeset. One day later, the developer drafts documentation on how the codebase can be configured with different network device names. He commits this new document in later changesets, in comments 13-16. By consulting this bug, the entire list of *what* changesets occured to handle the bug is readily available.

5 Integration Daemon

Inadequacies of existing systems called for a solution based on the abstraction of an integration daemon, as discussed next.

```
SCMBUG ID:   417
SCMBUG NOTE: Now using the name of the appropriate network device on
each system SigmaWatch runs. This name is set through a global
variable.
```

Figure 2: Documenting a fix for bug 417 during a `'cvs commit'`.

```
2004-08-21 Saturday 11:47  mkgnu

* src/host_node/userspace/server/readnet.c (1.9,
p_kpm_prior_to_bug424_fixes),
src/host_node/userspace/server/readnet.h.in (1.1,
p_kpm_prior_to_bug424_fixes), configure.in (1.33): SCMBUG ID:
417 SCMBUG NOTE:Now using the name of the appropriate network
device on each system SigmaWatch runs. This name is set
through a global variable.
```

Figure 3: Log comment entered in CVS for Bug 417, presented in a more readable Changelog format using `cvs2cl`. Through integration, this CVS entry is also documented in Bug 417, comment 10 of product Σ-Watch in Bugzilla, shown in Figure 4.

5.1 Public Internet Deployment

First, deployment over the public Internet was required to meet our development model. Members of our research group maintain individual SCM repositories on their personal laptops, to allow offline development. For our defect-tracking needs, a publicly available bug-tracking system is shared. Even though integration with bug-tracking is not possible when a user is working offline, it is still possible to produce the differences between a repository and a working copy (e.g. using `'cvs diff'`), and commit changesets at a later time. Hence, this method of development is encouraged. When problems arise and development of personal research prototypes stalls, pointing each other to a bug-report, for help, is beneficial. By describing a problem in a public defect-tracker, rather than a defect-tracker running on personal laptops, easy accessibility and quick resolution of defects are facilitated.

We also collaborate with researchers from different labs and research organizations on jointly funded projects, and do not wish to permit them access in our LAN using a VPN. Similarly, the security policy of collaborators sometimes does not grant us access to their internal resources. Using a public defect-tracker overcomes this problem, and reduces the administrative overhead of managing separate, private defect-trackers per project or organization. Experience has shown that limited labor and system administration expertise resources exist in an academic environment.

5.1.1 Minimizing exposure

Bugzilla, our defect-tracker of choice, uses MySQL as a database backend. Inserting log messages and version numbers in Bugzilla requires accessing the database. Opening the MySQL TCP port over the public Internet is a serious security risk, since it exposes access to other applications running on the database system. By using a separate integration daemon, and keeping this TCP port closed, only an integration interface is exposed.

5.1.2 Stand-alone backends

Some bug-tracking systems may not use a database backend that listens to a TCP port at all, such as the Berkley DB backend. Another example is debbugs[16], the bug-tracking software of the popular Debian Linux distribution. It uses it's own file-based database and is accessible through an email gateway. The abstraction of an integration daemon permits access to such file-based, stand-alone backends from the Internet.

5.1.3 Integration Security

Our development model does not match the vision of existing integration systems, such as Perforce, ClearCase, CVSZilla, or McIntyre's and Quillan's integration work. These solutions are deployed in a local area network environment and directly connect to the bug-tracking database. The general assumption is that no malicious users exist in the local network. Since the integration daemon listens on a public port, it cannot be assumed that all integration requests will originate

```
------- Additional Comment #10 From Kristis Makris  2004-08-21 11:47 -------

Now using the name of the appropriate network device on each system
SigmaWatch runs. This name is set through a global variable.



Affected files:
---------------
1.8 --> 1.9 system/src/host_node/userspace/server/readnet.c
NONE --> 1.1 system/src/host_node/userspace/server/readnet.h.in
```

Figure 4: Bug 417, comment 10 of product Σ-Watch in Bugzilla. The reason *why* this changeset occured was documented 2 days earlier in comment 6, shown in Figure 5.

```
 ------- Additional Comment #6 From Kristis Makris  2004-08-19 12:16 -------

The module that collects network information consults the hardcoded
"eth0" network device, which does not exist in PDAs. A "wlan0" should
be used in them instead, since they have wireless access; not
ethernet.
```

Figure 5: Root of the problem documented in bug 417, comment 6 of product Σ-Watch in Bugzilla. The problem is not documented in CVS.

from trusted developers of our lab.

The integration communication protocol includes the SCM username of the developer committing a changeset. This information is needed to identify the developer originating the software change. It is also needed to map to the username of the developer in the bug-tracking system. However, it is possible for an attacker to produce phony integration requests with properly matched usernames, and either pollute the bug-tracking system with misleading comments or add/delete product version numbers. As a remedy, public key authentication can be used to confirm the identity of integration requests in communications between the glue and the daemon.

5.2 Public Integration Interface

There's no public interface that system integrators can use to communicate with a bug-tracking system. Integration efforts currently duplicate functionality implemented in the codebase of the bug-tracking system. CVSZilla and Quillan's integration work followed this approach. When the database schema is modified in the next release of the bug-tracking system, the integration breaks. All SCM repositories must have their glue updated for the new bug-tracking system in order to proceed.

Instead, functionality from the bug-tracking system's codebase is directly reused. For example, a variable in the integration daemon configuration file points to the path were the Perl libraries distributed with the Bugzilla codebase are deployed. The Bugzilla-specific module of the integration daemon issues calls to this library. When the Bugzilla instance is upgraded, the new Bugzilla codebase includes functionality already updated by the bug-tracker's developers to match it's database schema. One only needs to point this path variable to the location of the new source distribution and restart the integration daemon. Therefore, an upgrade of the bug-tracking system does not require upgrading the glue in each SCM repository using this integration. Additionally, if a user migrates from Bugzilla to a different defect-tracking system she only has to specify the name of the new system in the daemon configuration file, and restart the daemon.

The bug-tracking system's codebase does not always provide all the functionality needed by the verification logic. For example, some queries specific to Bugzilla's database schema had to be implemented. Informing bug-tracker developers of such integration-related queries is important in providing generic verification logic. The Bugzilla developers are now planning[19] to use XML-RPC to define a public, HTTP-based interface for integration with 3rd party tools, which will include all the functionality needed by Scmbug.

Implementation of a similar interface is also missing in other bug-tracking systems, such as MantisBT and AntHill. Still, integration with other bug-tracking systems is desirable. Until other systems implement a similar interface, the daemon abstraction serves as a mediator, integration interface, where custom queries can be implemented.

5.3 Lack of SCM Integration Support in Bugtrackers

Free bug-tracking systems, such as Bugzilla, still lack support for SCM integration in their database schema. For example, integration work must be able to match the username used in the SCM system with the username used in the bug-tracking system to enforce a valid bug owner check, as described in Section 7.4. A bug-tracking system should provide space in it's database for each user's SCM username. Since this support is missing in some defect-trackers, the need for username mapping is satisfied by the integration daemon.

5.4 SCM System Limitations

The abstraction of an integration daemon also makes it possible to develop solutions to some known integration problems of existing SCM systems.

For example, CVS lacks atomic transactions. As a side-effect, when the same log message is used to commit files in two separate directories, two integration actions occur using the same log message. Duplicate log messages are then entered in the bug-tracking system. Coming back to the integration example of Section 4, Figure 3 shows that revision 1.33 of the file `system/configure.in` was produced during the example changeset. However, Figure 4 is missing a corresponding entry for this modification in the affected files list. This change was documented in comment 11 with a duplicate log message.

It is possible to solve this problem by caching integration requests at the daemon for a configurable, short time interval. Reception of another integration request with the same log message indicates that the transaction should have been atomic. The affected files list can be merged, and the log message inserted in the bug tracking system only once.

6 SCM Hooks

In modern version control systems, certain SCM events can trigger scripts that perform additional work handling the event. As a *common denominator*, SCM systems are expected today to offer hooks on the following events:

- `pre-commit`. Used to verify if the commit should proceed.
- `post-commit`. Used to integrate a commit log message with a bug-tracking system. Invoked only if `pre-commit` succeeded.
- `pre-label`. Used to verify if a creation of a tag or branch should proceed.
- `post-label`. Used to integrate the tag or branch name with a bug-tracking system. Invoked only if `pre-label` succeeded.

Scmbug installs scripts in an SCM repository which are executed as hooks for each event. After information about each event is collected, integration proceeds through the SCM glue logic. Integration of SCM systems that do not provide these hooks is not possible.

Subversion does not provide the `pre-label` and `post-label` hooks. Tags and branches are created using the `'svn copy'` command in predefined directories named `/tags` and `/branches`[13]. They are later committed with a regular `'svn commit'`. Nevertheless, it is still possible to detect during a `pre-commit` event if a tag or branch action is underway. If the commit activity indicates that a new subdirectory is created under `/tags` or `/branches`, then the transaction corresponds to a labeling action.

7 Integration Policies

Central to the behavior of the common glue logic is the notion of integration policies. It is essential that the glue prevents developers from describing erroneous integration actions. At the same time, the capability to match multiple development models is equally important. A configuration file stored in the SCM system controls the overall behavior of the integration glue, as described in the next sections.

7.1 Enabled Integration

The integration can be disabled at any time, by changing the value of a flag in the glue configuration file. Hence, the SCM system can easily back-out from using this integration.

7.2 Presence of Distinct Bug IDs

SCM systems prompt a developer with a log message template prior to committing a changeset. In order to integrate the log message of a changeset

```
SCMBUG ID:
SCMBUG NOTE:
```

Figure 6: Scmbug log message template. Essential in determining the bug against which a changeset is commited.

with a bug id in the bug-tracking system, a bug id must be included in a parseable format in the log message. Thus, Scmbug expects the predefined template shown in Figure 6 to be filled-in during a commit action.

Given a log message in this format, the glue logic verifies that one or more bug ids were supplied. Accepting multiple bug ids is required when a changeset fixes a collection of defects.

Additionally, the glue verifies that the ids supplied are unique. A duplicate bug id is most likely an indication that a developer typed the wrong bug id.

7.3 Valid Log Message Size

Project managers and SCM repository administrators often have to convince lazy programmers that there is value in typing a detailed log message during commits, and in SCM in general. An optional policy that requires the log message to meet a configurable, minimum message size is used to address this problem. Messages with a size less than the minimum cause an SCM commit action to be rejected and force a programmer to recommit with increased log comment verbosity.

Simply printing a warning when a small log message is entered and accepting the commit is not enough. A small, incomplete, misleading, or practically useless log comment can cause great grief when a bug has creeped in the software. Such inadequate documentation on a changeset introducing the bug can send a developer in a laborious SCM history cross-examination and bug-hunting trip.

7.4 Valid Bug Owner

Formal defect-tracking processes often define a Change Control Board (CCB)[14] or project manager that dispositions bug ids to developers based on the components they touch. In such settings, developers are not expected to commit changesets that touch bugs assigned to other developers. A policy in Scmbug verifies that the user issuing an integration action is the owner of the bug id specified in the bug-tracking system. This check ensures developers don't step on each other's assigned work.

7.5 Open Bug State

Even more important than valid bug owner checks is the capability to verify that a changeset is committed against a bug id set in an *open* state. For example, committing against a bug id that has been already resolved, marked either `FIXED` or `INVALID` (these are some resolution states in Bugzilla), would be wrong. Such verification checks alarm the developer that somebody else worked on the defect at hand already. Another example would be committing against a bug marked as `NEW` or `UNCONFIRMED`, where the CCB has not yet assigned the bug id to a developer. Examples of valid, *open* states would be `ASSIGNED` and `REOPENED`. This verification check ensures that a formal bug dispositioning process is followed.

7.6 Valid Product Name

A product name, specified in the glue configuration file, is transmitted to the integration daemon during an integration action. If the supplied bug id is associated with the specified product name in the bug-tracking system, the integration action proceeds. This verification check often captures cases where a developer enters the wrong bug id in a log message in an organization actively developing multiple products at the same time, such as Mozilla.org, and the GNOME and KDE projects.

This check assumes that an SCM system is used to host a single product. In some development models this may not be true. For example, a contracting company may choose to maintain documentation of multiple contracts in the same SCM system, while assigning different product names for them in the bug-tracking system. This approach reduces the administrative overhead of setting multiple SCM repositories. The valid product name policy also permits specification of multiple product names, to match such an alternative development model.

7.7 Convention-based Labeling

As software evolves through experimental, stable, or fork stages, the codebase may be labeled accordingly using the SCM system. Common labeling needs are:

- `Releases`. The codebase is tagged with a name indicating that it has reached a stable state, justifying a release point.

```
names => [
# Convention for official releases.
# For example:
# SCMBUG_RELEASE_0-2-7
'^.+?_RELEASE_[0-9]+-[0-9]+-[0-9]+$',

# Convention for development builds.
# For example:
# SCMBUG_BUILD_28_added_policies
'^.+?_BUILD_[0-9]+_.+$',

# Convention for branches.
# For example:
# b_glue_side_policies
'^b_.+$',

# Convention for private developer
# tags. Uses the developer's initials
# (either 2 or 3). For example:
# p_kpm_pre_bug353_fixes
'^p_[a-zA-Z][a-zA-Z]?[a-zA-Z]_.+$'
]
```

Figure 7: Label name convention examples. A list of regular expressions defines acceptable label names for releases, developer builds, forks and private labels.

- `Developer builds`. The codebase reached a developer milestone.
- `Forks`. A stable release will have important bug fixes supported in a separate branch. Forks may also be used to to implement experimental features without disrupting mainline development.
- `Private labels`. A developer labels the codebase either prior to or after introducing changesets that may introduce significant regression. The significance of the label is meaningful only to the developer.

In support of each of these labeling categories, a policy is introduced that ensures the label name used matches a configurable format defined as a regular expression. Figure 7 shows examples of label naming conventions that match these categories.

After a label name passes the convention check, it is entered in the bug-tracking system as an available version number on the specified product. The naming policy ensures developers apply a uniform labeling scheme. It also permits 3rd party tools to parse the available versions of a product in a consistent manner.

The convention-based labeling policy can be complemented by a role-based policy. A list of SCM usernames, in the form of a regular expression, authorized to create labels from each category could be specified. For example, only the release manager of a product should label releases and create forks, a group of high-ranked developers should label developer builds, and all developers should be encouraged to create private labels.

8 VDD Generator

Release management theory recommends software releases be paired with a document describing the changes since the previous release. Producing this document directly out of the SCM system, either using an automated tool, such as `cvs2cl` for CVS, or the SCM system itself, such as the `'svn log -r <rev_old>:<rev_new>'` command for Subversion, is not adequate. Changelog information derived strictly out of the SCM system is overly detailed. It describes software changesets at a lower, developer level, and is of little value to a user interested simply in a summary of added features. Moreover, when multiple changesets are committed in response to a defect such a Changelog document becomes lengthy. It takes considerable time to follow the history of changes and decipher if, or how, a defect was corrected.

It is more appropriate to pair such Changelog documents with a high-level summary of defects stored in the bug-tracking system. Summary fields are common in bugtrackers, but automating generation of this document involves determining exactly which bugs were worked on between releases. Identifying the date a release was labeled is a required first step (CVS does not support this), but it is not enough without integration of SCM with bug-tracking. Since the integration inserts log messages in the bug-tracking system, and the bug-tracker dates them, a list of bug ids that were worked on can be compiled simply by performing a date-range search on a product.

The bug-tracking system can then be requeried to report for each bug id not only the summary, but additional useful information. For example, it can report the resolution status(e.g. `FIXED`, `INVALID`), the bug owner(publicly attributing credit to the developer), resolution date, severity, priority, etc.

This report can also reflect decisions of the development team which are not documented in the SCM logs, such as choosing to not add a feature, resolving it as `WONTFIX`. It may also display bugs that were added in the period between releases but not worked yet, alerting users of newly discovered defects.

A tool that can produce such a version description document is under active development.

9 Early Experience

Scmbug has been deployed in our lab since March 2004 for integration of various research prototypes. In hindsight, we were late in developing a verbose logging mechanism in the integration daemon. Project deadlines and rapid development of Scmbug itself discouraged us from continuously upgrading to the latest version. We are unable to provide statistical information illustrating the frequency of developer integration errors that were caught by the verification policies.

9.1 Integration Upgrades

Seamlessly upgrading the integration work is not straight-forward. Between release 0.0.8 and 0.1.0 of Scmbug, the communication protocol between the glue and the daemon was altered. Upgrading the glue suddenly became a multi-stage process: (a) disable the glue, (b) install a newer glue release, (c) restart the integration daemon, (d) enable the glue. Disabling and enabling the glue was required to accept integration requests from a glueing codebase that matched the communication protocol understood by the integration daemon. It was also required for step (b) to succeed without communication with the daemon. Even though an automated glue installation and upgrading tool was employed, manual intervention was required on step (c) on the machine running the integration daemon. An important disadvantage is that upgrading multiple repositories requires all updated ones to remain without Scmbug integration until all repositories finish steps (a) and (b), before (c) is carried out. However, additional scripting by a system administrator could further automate this process.

After rapid development of our integration work, including rearranging of the Perl packaging structure installed in an SCM repository, the multi-stage process was proven to be defective in its implementation. Upgrading from release 0.1.1 to 0.3.1 revealed that the glue libraries used by the generic SCM hook processing script had changed package names and path, and the hooks were failing to carry out step (b). The solution to this problem was to completely remove the hooks rather than disable them through our configuration file.

Another design problem is that older glue installations are not preserved in the SCM repository. If a newer Scmbug release contains grave defects, one may not easily revert back to an older revision of the integration work. For example, CVS versions the hooks themselves. Committing a disabling hook, which is a hook that no longer invokes the glue processing script, still requires the original hook invoked for the last time. If this hook is defective, committing will fail. The defective hook will not be disabled, reaching a dead-end in upgrading the hook. In CVS one must then locally modify the repository to disable the hooks using separate RCS commands (RCS is the underlying database store of CVS). The installation mechanism can be enhanced to install permanently in an SCM repository every release of the integration glue, and allow switching between glue releases. This installation mechanism will need to be run locally on the machine hosting an SCM repository, to solve CVS's dead-end hooks problem. The integration daemon can also be enhanced to dynamically support older communication protocols.

9.2 Case Studies

Three undergraduate students in a microprocessor systems hardware course worked on one of our research prototypes for 16 weeks. They were introduced to our lab's development model, and a separate CVS branch was created for them to commit their changesets. The students worked on a total of 35 bugs. Being new to the concept of SCM, they occasionally checked out the main development line (often referred to as `HEAD`) instead of their personal branch. Consequently, they committed some changesets in `HEAD` instead of their personal branch. A policy for tuning fine-grained, branch-level, permission checks can solve this problem. For example, committing and labeling activities could be limited only to ourselves in `HEAD`, and students could be confined to committing only in their branch. Other developers using Scmbug also requested such a policy.

Development of another research prototype was integrated with Scmbug. Two developers experienced in the concept of SCM worked in the same lab, on 219 bugs, using CVS, for 15 weeks. Empirically, we can report that the most frequent verification check failed was a valid log message size check for 50 characters. This check correctly reminded the developers that they should be more verbose in documenting changeset history. The valid bug owner check was the second most frequent to fail. This was a result of miscommunication between developers. We speculate that this check may fail more often in open-source projects deploying Scmbug: the geographically distributed nature of developers in such a setting prohibits them from face-to-face communication on a daily basis. This check, along with the valid product name check, also failed due to typing errors while entering the bug id in the log message template. Finally, the least frequent verification check to fail was the open bug state check. This resulted from developer error in using TortoiseCVS[12], a GUI CVS client, to commit changesets. This client caches the most recent log messages. Developers often selected one of the previous messages in order to bring up the default log message template and type in a new message. After clearing out the old log message and typing a new one, developers occasionally forgot to change the bug id present in the cached log message. They instead attempted to commit against resolved bugs.

9.3 Just In Time Integration

Within five months of making Scmbug publicly available, we received over 6300 hits(49 per day) on the project's webpage[20]. The software distribution has been downloaded over 3200 times(26 per day). Other system integrators, including members of the Bugzilla integration team, discussed our design and recommended policy features which we implemented. Users reported deployment of Scmbug to successfully integrate both CVS and Subversion with Bugzilla, an activity that is largely simplified via an automatic installation script. It is clear that this integration tool fills a significant void in the open-source development community.

Why wasn't such an integration system built already? We were unable to find papers documenting the benefits of integrating SCM with bug-tracking, or proposing a similar solution. The Scmbug design is very flexible, and at the same time extremely simple. While the policy mechanism proposed is remarkably useful, it is not difficult to implement.

One reason might be that limited time is often available by developers for work on open-source projects. Sometimes, good-enough, quick hacks are easier to implement and preferred, rather than a full-blown, well-designed solution. For example, a common limitation of other systems integrating CVS with bug-tracking resulted from the inadequate mechanism CVS uses to provide the list of affected files in a commit trigger. A processing script using a single regular expression to parse these arguments gets confused if the filenames contain either commas or whitespaces. Scmbug handles this issue by employing a stateful parser, marginalizing the possibility of getting confused. The single regular expression method implemented in McIntyre's `bugzilla-watcher` script uses only 4 lines of Perl code for parsing filenames. Our parser was implemented in 145 lines of Perl code, and required significantly more time to develop.

A plethora of open-source SCM systems emerged in the past two years as alternatives to the currently dominant CVS. Subversion and OpenCM overcome various limitations of CVS and are still in active development. Arch was specifically designed for the distributed development needs of open source projects, such as the development model followed by the Linux kernel, and is still enhancing. Monotone is still in an alpha state and has been released only a year ago. We theorize that as these tools are still progressing to a stable state, integration with bug-tracking has yet to become a priority in their to-do list, and hence has not been addressed.

It is astonishing that mature, high-profile, public open-source software development websites still lack integration between their SCM and bug-tracking services. Some examples are Sourceforge (hosting over 93,100 projects, 983,900 users, in service for 5 years) and GNU Savannah (over 2,200 projects, 32,200 users, in service for 4 years). Our solution has been in high demand for a long time.

10 On-Going Work

OpenCM can commit changesets in disconnected mode[23], a feature also explored by Subversion developers. Laptop users can cache a repository and work offline. When connectivity is restored, they can synchronize their working set and resolve conflicts with the main repository. Scmbug could be improved to support a disconnected mode of integration. An integration daemon proxy, running on the user's laptop, could cache bug-tracking metadata required for policy verification checks, such as the list of bug ids, their state, bug owner, etc. All integration activity generated by disconnected commits could also be cached and synchronized later with the bug-tracker.

The star-topology development model assumed by Arch increases the significance of integrating branch and repository names. In this model, changesets may be produced by distributed SCM repositories, maintained by separate development teams. For example, a public Bugzilla instance tracks[22] defects in the Linux Kernel. Independent developers and companies maintaining private forks of the kernel post patches to bugs reported by anyone. However, they do not always report the name and branch of their repository fixing the bug. A user inspecting a bug report is uncertain which public tree includes the fix. Capturing a `<branch, repository name>` pair, which uniquely identifies the source of a changeset, could be supported by Scmbug. Providing generic support for this integration, would require all SCM systems to pass as arguments in integration hooks this information. The dominant CVS system, does not provide this information.

Improvements in SCM and bug-tracking systems are critical for a successful, generic integration solution. These are: (a) Section 5.2's public integration interface by bug-tracking systems, (b) the SCM username support of Section 5.3 by bug-tracking systems, (c) capturing the date a release was labeled by SCM systems, as mentioned in Section 8, (d) supplying a default log template in SCM systems, required by Section 7.2 (Subversion does not support this), the SCM hooks of Section 6 and (e) reporting in SCM hooks the `<branch, repository name>` pair just mentionied.

On-going work on Scmbug includes: (a) the public key authentication scheme of Section 5.1.3, (b) the role-based policy described in Section 7.7, (c) the fine-grained, branch-level permission policy mentioned in Section 9.2, (d) the VDD generator of Section 8, and (e) the improved upgrading mechanism of Section 9.1.

11 Conclusion

We presented Scmbug, a system offering policy-based integration of software configuration management with bug-tracking. Integration of SCM with bug-tracking improves the traceability of software changesets, the quality of documentation in defect reports, and quality of release documents.

Scmbug integrates activities such as committing software changesets and labeling software releases. The integration policies can be tuned to match multiple development models and provide synchronous verification checks. The design is flexible enough to support any SCM system with any bug-tracking system, can be deployed over the public Internet, improves the quality of release documentation, and can overcome limitations of existing systems. Finally, improvements in SCM and bug-tracking systems that are critical for a successful, generic integration solution are suggested.

12 Acknowledgments

Development of this system and paper benefited tremendously from insight on SCM and bug-tracking issues, along with invaluable feedback, from John C. Quillan, Mark S. Reibert, Dave Swegen, and the Bugzilla and Subversion developers. We regret Quillan's integration was never publicly released. We are convinced his work could have evolved to what Scmbug is today, and would have saved us considerable research effort.

We would also like to thank the existing Scmbug users for their increasing interest, testing, and continuous support of this integration system, especially Jay Glanville for his fruitful feedback.

13 Availability

Scmbug is free software, licensed under the GNU General Public License (GPL). It is available from `http://freshmeat.net/projects/scmbug`.

References

[1] AntHill. http://freshmeat.net/projects/anthill/, 2004.

[2] Bugzilla. http://www.bugzilla.org, 2004.

[3] ClearCase. http://www-306.ibm.com/software/awdtools/clearcase, 2004.

[4] MantisBT. http://freshmeat.net/projects/mantis/, 2004.

[5] Monotone. http://freshmeat.net/projects/monotone/, 2004.

[6] OpenCM. http://freshmeat.net/projects/opencm/, 2004.

[7] Perforce. http://freshmeat.net/projects/perforce/, 2004.

[8] Perforce Defect Tracking Integration. http://freshmeat.net/projects/p4dti/, 2004.

[9] Subversion. http://subversion.tigris.org, 2004.

[10] The Bugzilla Guide. http://www.bugzilla.org/docs/2.18/html/, 2004.

[11] The Concurrent Versions System (CVS). http://www.cvshome.org, 2004.

[12] TortoiseCVS. http://www.tortoisecvs.org, 2004.

[13] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, 2004.

[14] Susan Dart. Concepts in configuration management systems. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 1–18, 1999.

[15] Tony Garnock-Jones. CVSZilla. http://homepages.kcbbs.gen.nz/˜tonyg/, 2000.

[16] Ian Jackson. debbugs. http://www.chiark.greenend.org.uk/ ian/debbugs/, 1994.

[17] Gregor Joeris. Change management needs integrated process and configuration management. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 125–141. Springer–Verlag, 1997.

[18] Tom Lord. Arch revision control system. http://freshmeat.net/projects/archrevctl/, 2004.

[19] Kristis Makris. Provide an interface for SCM integration. Bugzilla Bugzilla, ID 255400, 2004.

[20] Kristis Makris. Scmbug. http://freshmeat.net/projects/scmbug/, 2004.

[21] Steve McIntyre. CVS/Bugzilla integration. http://www.einval.com/˜steve/software/cvs-bugzilla/, 2004.

[22] OSDL. Linux Kernel Bug Tracker. http://bugme.osdl.org/, 2004.

[23] Jonathan Shapiro and John Vanderburgh. CPCMS: A Configuration Management System Based on Cryptographic Names. In *2002 USENIX Annual Technical Conference, FREENIX Track*, 2002.

Linux Physical Memory Analysis

Paul Movall
International Business Machines Corporation
3605 Highway 52N, Rochester, MN

Ward Nelson
International Business Machines Corporation
3605 Highway 52N, Rochester, MN

Shaun Wetzstein
International Business Machines Corporation
3605 Highway 52N, Rochester, MN

Abstract

We present a tool suite for analysis of physical memory usage within the Linux kernel environment. This tool suite can be used to collect and analyze how the physical memory within a Linux environment is being used.

1. Introduction

Embedded subsystems are common in today's computer systems. These embedded subsystems range from the very simple to the very complex. In such embedded systems, memory is scarce and swap is non-existent. When adapting Linux for use in this environment, we needed to keep a close eye on physical memory usage.

When working on such a subsystem, we experienced various out of memory situations, ranging from significant application performance problems due to thrashing of executable pages to the killing of selected processes by the automated out of memory handling of the kernel.

After numerous Internet searches, we determined that there was no publicly available tool suite to analyze physical memory usage in real time on a running system using the Linux kernel. There are many tools, such as mpatrol [1] or memprof [2], which can be used for memory analysis. These tools focus on dynamically allocated virtual memory usage within a particular process. However our requirement was for systematic view physical memory usage across all usage types.

A significant amount of this information can alo be found in the /proc filesystem provided by the kernel. These /proc entries can be used to obtain many statistics including several related to memory usage. For example, /proc/<pid>/maps can be used to display a process' virtual memory map. Likewise, the contents of /proc/<pid>/status can be used to retreive statistics about virtual memory usage as well as the Resident Set Size (RSS).

In most cases, this process level detail is sufficient to analyze memory usage. In systems that do not have backing store, more details are often needed to analyze memory usage. For example, it's useful to know which pages in a process' address map are resident, not just how many. This information can be used to get some clues on the usage of a shared library. Also, since this information is process based, there is no way to tell if the resident pages are shared between processes. Yet this sharing of pages is a very important optimization performed by the kernel that can not be tracked with the existing tools.

We researched the Linux VMM from [3] and started the process of developing a tool suite to analyze the use of physical memory within a subsystem. This included the usage of memory by the kernel and user space applications. This paper focuses on the analysis of user space applications. Kernel usage was determined using the slab information.

2. Architecture

The basic requirement of the tool suite is to be able to determine physical memory usage in real time of an operational system with minimal impact to that system. In addition, in the case where a kernel panic did occur and a dump was available, the tool suite should be able to extract the pertinent information from the dump and provide the same analysis capability based off of the dump.

2.1. Design Goals

The design goals were to keep it simple, keep it small, and provide useful information in a very quick turn around time.

The tool suite was divided into data collection and data analysis tools. This allows each tool to be customized to the particular environment in which it must run. It also allows for a separation of the functionality and staging of the deliveries. In fact the initial delivery of the tool set contained only the data collection tools. The data was analyzed manually within a spreadsheet. The analysis tools were added later to help automate the analysis process.

2.2. Implementation

As mentioned previously, the tool suite is split into a data collection tool and a data analysis tool. The following sections describe each tool independently.

2.2.1. Data Collection

The data collection tool started out with two versions. The first version was a loadable kernel module that was installed into the kernel and provided a user interface to extract the data. It used the global internal kernel data structures of the VMM to collect the various information of virtual and physical memory usage. The internal VMM structures need to be collected as the virtual memory of a process is created at the request of the process, but the physical memory is only assigned when it has been accessed. Without the VMM structures, there is no way to determine which virtual pages have physical pages associated with them.

The second version was a user space application that probed the kernel memory through the /dev/kmem device. This version is more difficult to build and maintain than the module as kernel structures are being used in user space. Also, the impact of the user application on the running system was larger than that of the kernel module. Lastly, there was some function that was not possible to implement with the user space application. Due to these factors, only the kernel module is support for data collection on a running system.

A third version of the data collection is available for the `crash` utility as described in [4]. The tool suite provides a data collection module that can extend the `crash` utility when the `mcore` form of the Linux kernel dump is used. This provides data in the same format as the kernel module so that the same analysis tools can be used.

All versions of the collection tools collect the same information in the same format. This is required to be able to feed into the same analysis tool. The data starts out with the process information. This includes the process identifier (PID), the command used to start the process, the value of the pointer to the memory management (MM) structures, page fault numbers, and used time counters.

These last numbers have recently been added. The analysis tool can parse them but does not currently perform any analysis on them. The page fault numbers have been added to get an indication of how many pages faults happen over a period of time. That metric can be used as an indication of approaching the point of thrashing.

After the process information is dumped, the data is dumped in a set of three values. The first part of this trio is the virtual memory area (VMA). This provides the virtual start address, length, and various flags. The second part of this trio is a summary of the page table entries (PTEs) that are associated to that VMA. The third part of the trio is the detailed mappings from virtual page to physical page that is provided by the PTE. There is also a set of flags with this information.

The output is in a terse format to minimize space and limit the performance and memory requirements to collect and store the data. It dumps this information for all Linux tasks, resulting in a requirement on the analysis tool to sort out thread from processes. This can be done using the MM pointers. Sample output of the collected data is provided later in this document and shown in Figure 1.

2.2.2. Data Analysis

The data analysis tool takes as input the output from the data collection tools and produces a variety of formatted output. In its simplest mode, it provides a comma separated value file that can be imported into a spreadsheet for further analysis manually. It can also provide graphic output, in the form of PostScript, of various metrics of physical and virtual memory usage.

The data analyzer parses information in the following forms.

- Basic process information, available in either physical or virtual memory views as PostScript or comma separated values, which includes:
    - total number of read-only and read/write pages assigned to a process,
    - number of pages assigned to the process stack,
    - number of pages assigned to global data areas (both .bss and .data),
    - number of pages assigned to heap space,
    - number of pages assigned to the application executable code,
    - number of pages in shared library global data (.data and .bss),
    - number of pages in shared library executable code;
- A view of the unique virtual memory areas (VMAs) within the process including virtual and physical sizes and the mapping of physical pages within the virtual space;
- A view of all unique executable entities (application code and shared libraries) coalesced across all processes that shows the virtual and physical sizes as well as the mapping of physical pages within the virtual space;
- For all unique executable entities (application code and shared libraries) a view of the count of physical pages for the executable (i.e., .text) and global data (i.e., .data and .bss) areas of each entity, available as a physical or virtual view as PostScript or comma separated values;
- Detection within a virtual memory area of single and multiple use pages (more on this later);
- The total number of physical pages used by user space applications;
- A dump of the physical page structures used within the kernel that are assigned to user space processes for further manual analysis.

Sample output of the various analyzed data is provided later in this document.

2.2.3. Operating Environments

The kernel module supports both PPC and Intel processor architectures. It uses only architecture independent structures, interfaces, and macros, and should be easily ported to other processor architectures.

The dump data extractor supports a PPC dump being analyzed on an Intel Linux environment. There are some architecture dependent structures used in this environment and therefore would be harder to port.

The data analysis tool supports Intel Linux and Intel Cygwin operating environments. It uses a simple text based selection mechanism and has no graphical user interface. It should port easily to other GNU C environments.

3. Usage

The following sections provide sample invocations and output of the various portions of the tool suite.

3.1. Data Collection

The primary collection tool is the kernel module. It can be installed using the insmod command.

```
insmod physmem_info_ppc.o
```

The output of the kernel module is extracted from a special character device. In the current implementation, this device is created using devfs. A device interface was used as the amount of data can be quite significant. It also minimized the complexity in the driver by not having to provide a simple character stream for the data. All of the buffering has been pushed out of the kernel and into user space. A simple mechanism to extract this information is to use the dd command. In order to keep the kernel module simple, the device actually blocks all of the task data into one read command. To determine the minimum read size, use the information in the following file.

```
/proc/physical_info/debug_task_mem
```

The content of this file will specify the maximum size of an output block. Using this information, the following command can be used to extract the physical per process usage.

```
dd   if=/dev/task_mem/0   of=<output
file> bs=<block size from /proc/phys-
ical_info/debug_task_mem>
```

Now the output will be located in <output file>. Since this information can be quite large, it is unlikely that the information will be able to be stored directly on the embedded system. In our usage, an NFS mount was used to store the output file.

Sample output of the collected data is shown in Figure 1. This example provides a good sample of the data. The task line starts with the T: and provides information on the kernel task. This data includes the command use to start the task, the process identifier, the pointer to the memory map, the CPU number that this process is associated with, page fault information, and usage times. The memory map pointer is used by the analysis tools to determine the difference between a pthread within a process (with the same memory map) and a unique process with the same name (different memory map). The other information, such as the page faults, can be used to detect thrashing. The time measurements could be used by performance analyzers to determine the usage patterns across various processes across time.

For each of the tasks, the virtual memory areas (VMAs) are listed and each starts with V:. Each VMA item shows the starting and ending virtual address of the area, the size in virtual, the size that is physically resident in memory, various flags, and the executable unit to which this VMA belongs. The various flags are critical to analyze the usage of the VMA. From the flags, it can be determined if the VMA is used for text, data, or stack areas. Also, the executable unit is used in the analysis. For

```
T: swapper , 0, 00000000, 0, 0, 0, 0, 0, 0, 39626
T: init , 1, c018c0a0, 0, 392597, 160, 69857, 174569, 8771, 30193
V: 0FE94000, 0FFB0FFF, 0011D000, 0005D000, 00000412, 00000075, /lib/libc-2.2.5.so
P: 93, 0, 93, 0, 93
M: 0FE94000=02876012, 22, 000420CC | 0FE95000=02849012, 22, 000420CC | 0FE96000=02848012, 22, 000420CC |
0FE97000=02847012, 22, 000420CC | 0FE98000=02846012, 22, 000420CC | 0FE99000=02845012, 22, 000420CC |
0FE9A000=02844012, 22, 000420CC | 0FE9B000=02843012, 22, 000420CC |
...
V: 0FFB1000, 0FFB3FFF, 00003000, 00000000, 00000402, 00000070, /lib/libc-2.2.5.so
P: 0, 0, 0, 0, 0
M:
V: 0FFB4000, 0FFC7FFF, 00014000, 00008000, 00000412, 00000077, /lib/libc-2.2.5.so
P: 8, 0, 5, 8, 0
M: 0FFC0000=02857092, 1, 00000054 | 0FFC1000=02856492, 1, 000000D4 | 0FFC2000=02855492, 1, 000000D4 |
0FFC3000=023E15D2, 1, 000020C4 | 0FFC4000=02870692, 1, 000020D4 | 0FFC5000=02853492, 1, 000000D4 |
0FFC6000=02ACA5D2, 1, 000020C4 | 0FFC7000=0225F7D2, 1, 000020C4 |
V: 0FFC8000, 0FFCCFFF, 00005000, 00004000, 00000412, 00000077, NULL
P: 4, 0, 3, 4, 0
M: 0FFC8000=02860692, 1, 000220D4 | 0FFC9000=02819492, 1, 000000D4 | 0FFCA000=02792092, 1, 00000054 |
0FFCB000=01D3E5D2, 1, 000020C4 |
...
T: keventd , 2, 00000000, 0, 0, 0, 0, 0, 0, 229
T: ksoftirq, 3, 00000000, 0, 0, 0, 0, 0, 0, 231
T: tiny_sys, 19, c018c2a0, 0, 14, 7, 0, 0, 762, 1971
V: 0FC3A000, 0FC44FFF, 0000B000, 00003000, 00000412, 00000075, /lib/libgcc_s.so.1
P: 3, 0, 3, 0, 3
M: 0FC3A000=025B3012, 15, 000420CC | 0FC3B000=025AD212, 15, 000420CC | 0FC44000=025B0212, 15, 000420CC |
V: 0FC45000, 0FC49FFF, 00005000, 00000000, 00000402, 00000070, /lib/libgcc_s.so.1
P: 0, 0, 0, 0, 0
M:
V: 0FC4A000, 0FC55FFF, 0000C000, 00002000, 00000412, 00000077, /lib/libgcc_s.so.1
P: 2, 0, 2, 2, 0
M: 0FC54000=025A5092, 1, 00000050 | 0FC55000=025AE292, 1, 00002050 |
...
```

Figure 1 Sample Data Collector Output

anonymous areas, the executable unit is NULL. This is typical for heap areas and bss areas.

Also with the VMA is the page table entry summary for that particular VMA. This is started with the P: key. The information in this stanza of the output is the number of user space pages, executable pages, read-only pages, single-use pages, and shared pages. All of these are in terms of physically resident pages.

For those VMAs with physically resident pages, the mapping from virtual to physical is also provided by the M: stanza. The information provided here includes the virtual address, the physical address, the usage count of this physical page, and various page flags. Note that these are separated with a unique character to ease parsing in the analyzer.

3.2. Post Processing / Analysis

Now that the data has been collected, it must be analyzed for meaningful usage. As discussed earlier in this paper, the analyzer is capable of performing 8 different types of analysis. Some of these have various forms including comma separated values, graphical in PostScript, or simple text. Each of these is described below and samples are provided.

3.2.1. Basic Information

This option provides basic summary information on an application (or process) basis. This information consists of the number of pages for the following:

- Stack
- Heap and global uninitialized data (.bss) for the application
- Global initialized data (.data) for the application

All of the above fields can be provided in both virtual pages and physical pages. The physical pages option provides only the actually resident page counts, whereas the virtual option provides the full virtual size requested. In addition in the virtual option, the graphical output has both physical and virtual indicators so that direct comparisons can be made.

An example of the comma separated value output for this option is provided in Figure 2. An example of the graphical output displaying this for both virtual and physical is provided in Figure 3. In this example, the tiny_sys application has over 800 pages of virtual address space and almost 300 pages of physically resident memory. Each of the various memory types is represented by a separate color on the graph.

3.2.2. Information per VMA

This option of the analyzer provides information on how a VMA is mapped to physical pages. This information can be used to determine how much of a library is used. It can also be used to arrange data areas within a VMA to make the accesses of these areas have better affinity. An example of this information is provided in Figure 4.

Some things to note in this example are the various usage counts of the pages. This is shown in the mapping as 1 through 9 for that number of physical mappings to that page, or as a '#' for 10 or more mappings to that physical page. To make the map readable, it is broken down into 64 pages per line. In the case that the VMA is not a 64 page multiple, spaces are used to indicate that no virtual space occupies that part of the map. For virtual pages that do not have a physical page, a period is used. In addition, the flags are decoded into a human readable form with the following mapping:

- 'S' is for stack;

```
Name, RW Count, RX Count, Total, Name, Stack, .Data, .Text, Heap, Lib .bss, Lib .data, Lib .text, Single, Multi
swapper, 0, 0, 0, swapper, 0, 0, 0, 0, 0, 0, 0, 0, 0
init, 24, 130, 154, init, 4, 1, 12, 4, 5, 10, 118, 0, 118
tiny_sys, 40, 243, 283, tiny_sys, 2, 1, 2, 2, 8, 27, 241, 0, 241
softdog, 20, 112, 132, softdog, 2, 1, 1, 3, 5, 9, 111, 0, 111
link-mon, 24, 132, 156, link-mon, 2, 1, 29, 8, 4, 9, 103, 0, 103
inetd, 24, 115, 139, inetd, 2, 1, 4, 5, 6, 10, 111, 0, 111
```

Figure 2 Sample Basic Output In Comma Separated Value Format

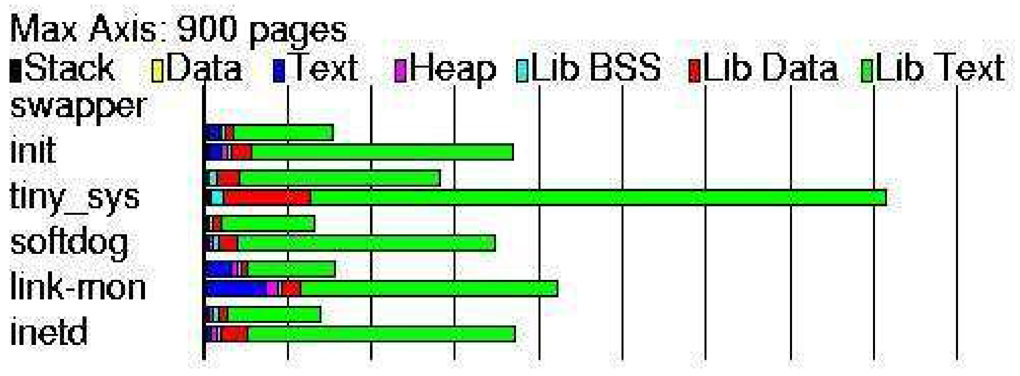


Figure 3 Sample Basic Output in Graphical Format

- 'X' is for association directly to the application and not to any of the shared libraries required by the application;
- 'r', 'w', 'x', for read, write, and execute permissions respectively;
- 'B' for .bss or heap data areas.

After the VMA mappings, the application summary of how each of the physical pages are used is also provided. These are all shown in this example.

3.2.3. Information for Executable Units Across All Processes

The previous information provided a view of each VMA for each particular process. This option provides similar information, but this time across all processes. This information can be used to determine the total usage of a shared library. This can be useful information for the amount of overhead associated with all shared libraries across the entire system.

An example of this information is show in Figure 5. Note that this is different than the previous example as this information is organized by executable unit rather than by process. Also note that the mapping of virtual pages to physical pages follows the same format as in the previous example, with a period meaning no physical mapping.

3.2.4. Total Memory Usage per Executable Unit

This output provides the information on the total memory usage of a particular executable unit. As with the per process view, this output is provided in both virtual memory and physical memory pages. This provides the total usage across all portions of the executable unit including code and global data. In this view, pages assigned to a particular process, such as stack and heap space, are not included. This option is to help analyze shared library usage of memory.

An example of this is shown in Figure 6. Again, this example provides both a virtual and a physical view of required and used pages. For example, libc requires just over 400 virtual pages and approximately 175 physical pages across all processes. The virtual requirement is split into approximately 25 pages of uninitialized global data, 100 pages of initialized data, and 275 pages of code. This data is also available in the comma separated value format suitable for import into a spread sheet.

3.2.5. Non-shared Pages in Shared Libraries

The next analyzer output shows the virtual memory areas that have both single use and shared use pages. If this VMA is used in another process, all of the single use pages are copied into each process instead of being shared. An example of this is shown in Figure 7. As shown in the example, the flags for the VM areas have read and execute permissions, indicating that these are

```
init:
  vsize:  285, single:    0, shared:   93, flags: ..r.x., 0X0FE94000, 0X0FFB0FFF, name: /lib/libc-2
[ ##############################...................####.#.........]
[ ..........###.#....####.##........####....##.#########.######...]
[ ................#####....##...##.......................#......##]
[ #......#..#.#.#...............................................#.]
[ ...............##.#####.#....                                   ]
  vsize:    3, single:    0, shared:    0, flags: ......, 0X0FFB1000, 0X0FFB3FFF, name: /lib/libc-2
[ ...                                                             ]
  vsize:   20, single:    8, shared:    0, flags: ..rwx., 0X0FFB4000, 0X0FFC7FFF, name: /lib/libc-2
[ ............11111111                                            ]
  vsize:    5, single:    4, shared:    0, flags: ..rwxB, 0X0FFC8000, 0X0FFCCFFF, name: NULL
[ 1111.                                                           ]
  vsize:    3, single:    0, shared:    2, flags: ..r.x., 0X0FFDD000, 0X0FFDFFFF, name: /lib/libmyinit
[ #3.                                                             ]
  vsize:   13, single:    0, shared:    0, flags: ......, 0X0FFE0000, 0X0FFECFFF, name: /lib/libmyinit
[ .............                                                   ]
  vsize:    3, single:    1, shared:    0, flags: ..rwx., 0X0FFED000, 0X0FFEFFFF, name: /lib/libmyinit
[ ..1                                                             ]
  vsize:   15, single:    0, shared:   12, flags: .Xr.x., 0X10000000, 0X1000EFFF, name: /bin/myinit
[ ..2222.22222222                                                 ]
  vsize:    1, single:    1, shared:    0, flags: .Xrwx., 0X1001E000, 0X1001EFFF, name: /bin/myinit
[ 1                                                               ]
  vsize:    7, single:    4, shared:    0, flags: .XrwxB, 0X1001F000, 0X10025FFF, name: NULL
[ 111.1..                                                         ]
  vsize:   23, single:    0, shared:   23, flags: ..r.x., 0X30000000, 0X30016FFF, name: /lib/ld-2
[ #######################                                         ]
  vsize:    1, single:    1, shared:    0, flags: ..rwx., 0X30026000, 0X30026FFF, name: /lib/ld-2
[ 1                                                               ]
  vsize:    1, single:    1, shared:    0, flags: ..rwxB, 0X30027000, 0X30027FFF, name: NULL
[ 1                                                               ]
  vsize:    4, single:    4, shared:    0, flags: S.rwx., 0X7FFFC000, 0X7FFFFFFF, name: NULL
[ 1111                                                            ]
Process "init" has 24 rw and 130 rx pages, (154 pages total)
         For [ init] : 4 stack, 1 data, 12 .text, 4 heap
         For libraries: 5 .bss, 10 .data, 118 .text
           0 are used only in this process, 118 are shared
```

Figure 4 Sample Output of VMA Information per Process

```
/lib/libc-2 is used 5 times with 285 virtual pages  and 109 resident pages and flags ..r.x. has the following map:
[ 555555555555555555555555555555..........111......5555.5.........]
[ ..........555.4....5333134........5443....34.4455555541555555.1.]
[ ................43444....22...5511..1..1..............44......45]
[ 5.2...35..3.3.5.............................................225.]
[ ...............54.53455141...                                   ]

/lib/libc-2 is used 5 times with 3 virtual pages  and 0 resident pages and flags ...... has the following map:
[ ...                                                             ]

/lib/libc-2 is used 5 times with 20 virtual pages  and 8 resident pages and flags ..rwx. has the following map:
[ ............55555555                                            ]

/lib/libc-2 is used 5 times with 5 virtual pages  and 5 resident pages and flags ..rwxB has the following map:
[ 55351                                                           ]

/lib/ld-2 is used 5 times with 23 virtual pages  and 23 resident pages and flags ..r.x. has the following map:
[ 55555555555555555555555                                         ]

/lib/ld-2 is used 5 times with 1 virtual pages  and 1 resident pages and flags ..rwx. has the following map:
[ 5                                                               ]

/lib/ld-2 is used 5 times with 1 virtual pages  and 1 resident pages and flags ..rwxB has the following map:
[ 5                                                               ]
```

Figure 5 Sample Output of VMA Information per Executable Unit

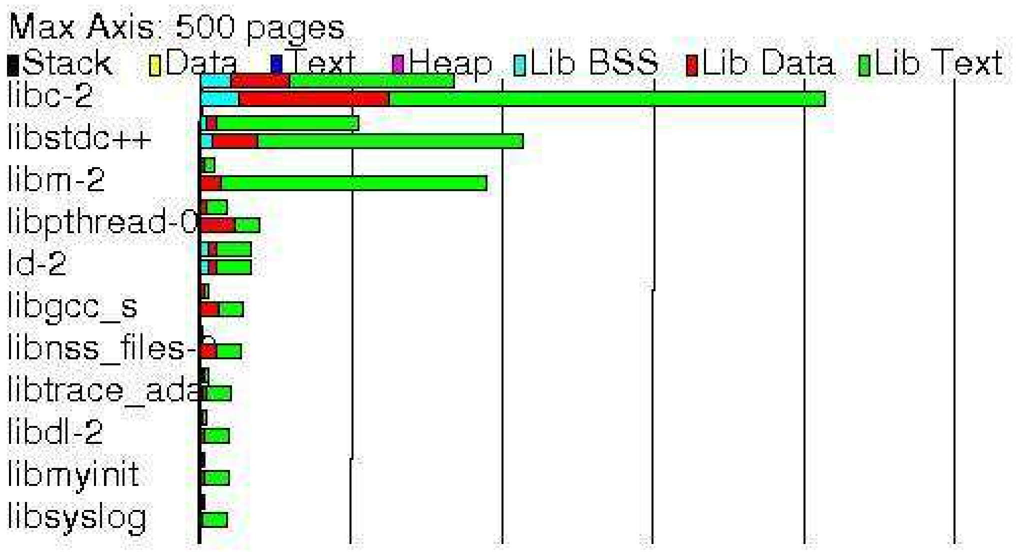


Figure 6 Sample Output of Total Memory Used per Executable Unit

```
Alert! VMA [ /lib/libc-2] in "link-mon" has 2 single usage and 6 shared usage, flags = 00000077
Alert! VMA [ NULL] in "link-mon" has 6 single usage and 2 shared usage, flags = 20001077
Alert! VMA [ NULL] in "link-mon" has 1 single usage and 1 shared usage, flags = 00000177
```

Figure 7 Sample Output for Non-shared Pages in Shared Libraries

```
User space pages: 439
```

Figure 8 Sample Output of Total Pages Used

3.2.6. Number of Physical Pages Used

In order to track the usage of physical pages from one code version to the next, the analyzer can simply output the number of physical pages used by user space processes. This information can then be easily tracked across each version of code. This is useful to insure that as changes are made and new function is added, that we do not approach the limit of physical memory. An example of the output is shown below in Figure 8.

In this example, 439 pages of physical memory where used by user space processes.

Note that all of the output has been in terms of pages. This keeps the entire tool suite, from the collection tools to the analyzer, to be architecture independent. To correlate to the actual number of bytes, the number of pages must be multiplied by the physical page size. For 32 bit PowerPC and Intel x86, the physical page size is 4kB.

3.2.7. Dump of Physical Page Structures

As shown in previous examples, the output of the analyzer can be provided in a format for further manual analysis. Typically this includes the CSV format so that the data can be imported into a spread sheet application. In this case, the output is in a human readable form for easier manual analysis.

The data provided for this option is the physical address, the PTE and VMA flags for that page, the symbolic representation of the VMA flags, the flags for the

```
Address       PTE Flag      VMA Flag      VMA Sym  Page Flags   Count   Owner
0X0018D000    0X00000012    0X00000875    ..r.x.   0X000420CC   35      /lib/ld-2
0X00190000    0X00000212    0X00000875    ..r.x.   0X000420CC   35      /lib/ld-2
0X00191000    0X00000212    0X00000875    ..r.x.   0X000420CC   35      /lib/ld-2
0X00192000    0X00000212    0X00000875    ..r.x.   0X000420CC   35      /lib/ld-2
0X00193000    0X00000212    0X00000875    ..r.x.   0X000420CC   35      /lib/ld-2
0X00452000    0X000007D2    0X00000177    S.rwx.   0X000020C4   1       init
0X00680000    0X00000612    0X00001875    .Xr.x.   0X000420CC   2       /bin/myinit
0X006E0000    0X00000412    0X00001875    .Xr.x.   0X000420CC   2       /bin/myinit
0X00746000    0X000005D2    0X00001877    .Xrwx.   0X00002048   1       /bin/busybox
0X008C7000    0X00000412    0X00001875    .Xr.x.   0X000420CC   2       /bin/myinit
0X00A46000    0X00000612    0X00001875    .Xr.x.   0X000420CC   2       /bin/myinit
0X00B06000    0X000005D2    0X20001077    .XrwxB   0X000000CC   1       /bin/myinit
0X00B5E000    0X00000612    0X00001875    .Xr.x.   0X000420CC   2       /bin/myinit
0X00BC1000    0X000005D2    0X20001077    .XrwxB   0X00002040   1       /bin/busybox
0X00BF9000    0X00000612    0X00001875    .Xr.x.   0X000420CC   2       /bin/myinit
0X00C03000    0X00000292    0X00000877    ..rwx.   0X00002054   1       /lib/ld-2
0X00C2C000    0X000007D2    0X00001877    .Xrwx.   0X000020CC   1       /bin/myinit
0X012A2000    0X000005D2    0X20001077    .XrwxB   0X00002040   1       /bin/busybox
0X01B6F000    0X000005D2    0X20001077    .XrwxB   0X000000CC   1       /bin/myinit
0X01BB2000    0X00000092    0X00000077    ..rwx.   0X000020D4   1       /lib/libc-2
0X01C04000    0X00000492    0X20001077    .XrwxB   0X000220D4   2       /bin/busybox
```

Figure 9 Sample Output of Page Structures

page within the `page struct`, the usage count of the page, and the executable unit to which the page is assigned. Details on the `page struct` can be found in [3]. This information can be analyzed to find any type of usage pattern or any other type of information that may become apparent upon inspection.

An example of this output is shown in Figure 9. In this example the first page shown is a text page (as noted by the flags of 'r.x') from /lib/ld-2 with a usage count of 35.

4. Results

We have uncovered numerous improper usage of memory in our testing using this tool suite. Below are some examples of the changes that have been made based on the output of this tool suite. None of these would have been easy to detect without this tool suite.

- Many shared libraries were not being compiled with the position independent code flag. This caused the text (i.e., executable code) pages to be copied uniquely into each process instead of being shared.

- The amount of overhead in physical pages for a process for each shared library it uses. While this latter fact would have been detected with easily available information (such as /proc/<pid>/maps), it was not obvious how this was contributing to the overall usage of physical memory.

- We have also been able to show that a number of processes leak memory. Using the tool suite, we were able to deduce an approximate leakage size. This would allow a developer to more quickly find their memory leak.

With the changes for position independent code, shared library repackaging, and a reduction in the number of linked shared libraries, we have been able to successfully reduce physical memory consumption. For example, the change for the position independent code flag on the compiled libraries saved over 4MB alone.

5. Summary

This tool suite identified many improper usages of physical memory. It has usage across any constrained memory system using Linux as its operating system.

For the future, the inkernel data collection tool needs to be ported to Linux 2.6. In addition, the analyzer should be enhanced to use the page fault numbers to detect trashing.

The source code for the tool suite is available from USENIX along with the transcript of this paper. The code is available as a tar.gz file that contains the source, Makefiles, and Doxygen control files.

6. References

[1] G. Roy, “mpatrol”, http://www.cbmamiga.demon.-co.uk/mpatrol/

[2] O. Taylor, “MemProf - Profiling and leak detection”, http://www.gnome.org/projects/memprof/

[3] M. Gorman, “Understanding the Linux Virtual Memory Manager”, http://www.skynet.ie/~mel/projects/vm/guide/pdf/understand.pdf, February 2003.

[4] Mission Critical Linux, “In Memory Core Dump”, http://oss.missioncriticallinux.com/projects/mcore/

Running virtualized native drivers in User Mode Linux [*]

V. Guffens G. Bastin
Centre for Systems Engineering and Applied Mechanics (CESAME)
Université Catholique de Louvain, Belgium
{*guffens,bastin*}*@auto.ucl.ac.be*

Abstract

A simulation infrastructure for wireless network emulation based on User Mode Linux and on the virtualisation of the hostap driver is proposed. The interconnection of these components is first described and the architecture of the resulting network emulator is explained. Two practical applications are then detailed : the testing of an implementation of the AODV routing protocol in a highly realistic environment and the study of the interactions between the hostap driver and the card it drives.

1 Introduction

User Mode Linux (UML [1]) has proven to be very useful for kernel debugging ([9], chap. 4), for the implementation of new functionalities in the kernel and as a testing and teaching tools. The Openswan developers, for instance, report in [7] the use of UML as a testing and development tools for their project. It is also used in [2] for network protocol testing with the VNUML project. UML is also used to implement web hosting solutions, honeypots and redundant services.

Some customs drivers exist in UML that provide, as an example, network connectivity. One of the existing Ethernet driver, for instance, works by opening a *tap* interface on the host side and by presenting this interface at the UML kernel side as the usual *net_device* structure with the suitable interfacing functions. From the point of view of the user land tools such as *ifconfig* or *ip*, this interface can therefore be manipulated as any other real Ethernet interface would.

However, these custom UML drivers are different from the native Linux drivers and the benefits mentioned above are therefore lost. Those benefits could nevertheless be recovered if the virtualisation process was carried on at a lower level. In this paper, we describe how this process can be successfully achieved by implementing a new bus, that we call *netbus*, which implements the functionalities found at the PCI level. This new bus allows for inserting native PCI Linux driver inside UML providing that some piece of code exists to emulate the hardware device that the driver is manipulating. To this end, the implementation of a software 802.11 card that can be operated by the *hostap* [6] driver is presented.

These software cards are then connected to each other trough a network server that we have developed (sources available at [3]) in QT/C++ and which provides a physical layer emulation as well as a graphical tool to represent the virtual machines in a 2-dimensional world (See figures later in the text). The advantages of such a system are as follows :

- It provides a complete wireless network emulator which is highly realistic and which can be used to test and develop new wireless related protocols. We report the implementation of the name resolution manet draft using this system in [4]. In this paper, we focus on the testing of the AODV multihop adhoc routing protocol. The implementation under investigation is the NIST kernel AODV module [5]. This implementation has been chosen because users from a wifi community [8] have reported random failures in the routing after many hours of operation. Diagnosing the cause of the failure on site might be a real nightmare and the solution is usually to reboot the device immediately. Reproducing the failure in the emulator might therefore be very valuable and help in development of such

[*]This paper presents research results of the Belgian Programme on Interuniversity Attraction Poles, initiated by the Belgian Federal Science Policy Office. The scientific responsibility rests with its author(s).

citizen networks.

- It can be used as a teaching tool to understand how a driver works.

In Section 2, the architecture of the emulator is first described and in Section 3 and 4 it is shown how it can be used to efficiently debug the AODV routing protocol and to help in understanding the interactions between a driver and the hardware it drives. Some related works are mentioned in Section 5 and the conclusion is given in Section 6

2 Emulator architecture

As described in the introduction, the main components used in the emulator are the User Mode Linux kernel code and a wireless driver. For this work, we chose the `hostap` driver [6] because it supports different kind of hardware and its hardware dependent code is therefore neatly separated. Furthermore, the `hostap` driver may be setup as an access point and supports software-based encryption. The advantages of the Linux kernel are, among others, availability of the source code, availability of advanced networking features, increasing use in embedded network devices and huge amount of networking related softwares. In the context of this paper, the availability of a stable User Mode port is of course essential.

Linking the `hostap` driver with the User Mode kernel requires the resolution of all symbols from the kernel's exported symbol table which is not possible as User Mode Linux does not contain any bus implementation. Unresolved symbols, whose names are self-explanatory are for instance : `pci_enable_device`, `writew` and `readw`.

The first component that had to be implemented was therefore a simple bus suitable to export the required symbols for the `hostap` driver to be inserted in the kernel. The modification of the `hostap` driver is minimal and consists in replacing unresolved symbols with the new ones, along with some minor modifications due to the simpler implementation of our bus compared to a standard `PCI` bus. We refer to this bus as "netbus" as its primary use is to connect a network device to UML. The functionalities of this bus, sending and receiving data as well as transmitting interrupt signals are implemented with TCP connections. Once the `hostap` driver is successfully linked into the kernel, it must of course act on a device which also has to be emulated.

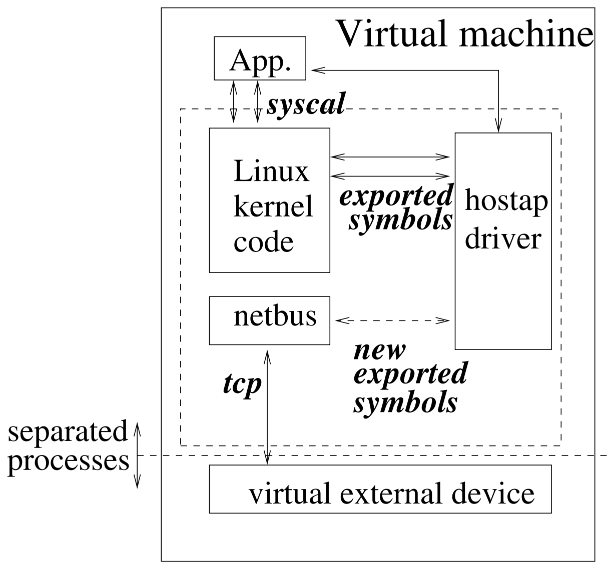


Figure 1: Interconnection of the driver with the UML kernel

2.1 Interconnection of the components

As any other device, our emulated wireless card has to be connected or "virtually plugged" into the UML via netbus. This situation is depicted in Fig. 1 showing the different interconnections existing between the components introduced above. The emulated wireless card runs as a different process, which is the emulator itself. This core program is written in QT/C++ and also implements the physical layer emulation as well as a visualisation system.

As mentioned above, netbus is built with TCP connections and the core emulator is therefore written as a TCP server.

The architecture of the emulator is depicted in Fig. 2 where it can be seen that the `hostap` driver is represented as a TCP client which is implemented on top of netbus. When the driver tries to register itself, it calls a function `netbus_register_device` which tries to bind to the emulator server socket. If the operation is successful, a new object implementing the virtual wireless card is instantiated in the emulator. The visualisation system displays an icon representing the mobile node with a surrounding circle which represents the reachability zone of the card (See more details later in the text).

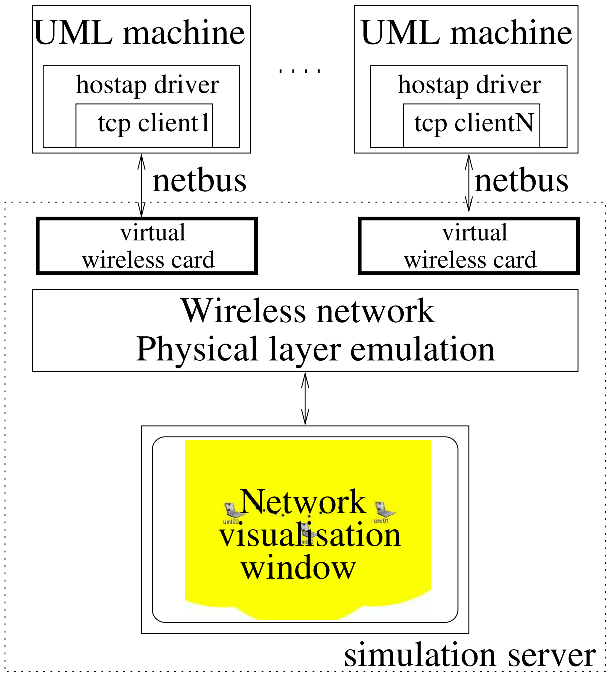


Figure 2: Architecture of our wireless emulator

2.2 Netbus implementation

As mentioned previously, the connection between the emulated card and the driver is realised with a TCP connection. Using TCP makes it possible to run multiple UML machines on different hosts. The simulation can therefore be distributed among multiple CPU's. The TCP protocol has been chosen for its reliability. Indeed, it is not desirable to introduce uncertainty in the delivery of interrupt and data as one would like to control in a deterministic fashion if packet drops occur or not.

In our prototype, two different TCP connections are used : one connection is used for data transmission while the other is only used for interrupt emission.

The client socket corresponding to the interrupt line at the UML side is configured in ASYNC mode and the associated SIGIO signal is registered in UML as an interrupt signal associated with the wireless card. The data line operates in blocking mode. This is necessary as, from the UML point of view, the read/write operation has to be "atomic" and must be completed entirely before the UML may continue to execute. Every command sent on the data line is therefore followed by a blocking `recv` call which waits for an acknowledgement from the card and synchronises the kernel with the card.

As packets are transmitted by the `hostap` driver by a series of `writew` command (programmed IO), the synchronisation mode described above induces a high latency in packet transmission. Therefore, an extra feature has been implemented in netbus which permits to send a block of "len" bytes of data in a single blocking operation. In some sense, this feature is analogous to a DMA in a real hardware.

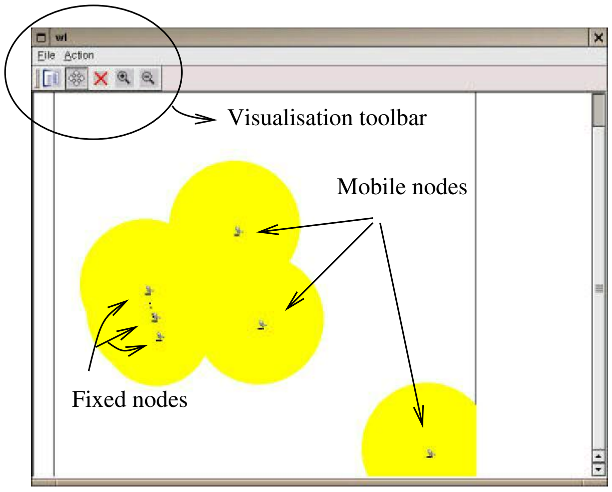


Figure 3: The setup used for testing the AODV protocol

3 A testbed environment

With the spreading of roof-top wifi networks, dedicated devices such as the *meshbox* and dedicated distributions such as *OpenAP* started to appear. Basically, these devices should be configured as AODV routers, installed at the right position and forgotten. Unfortunately, stability issues, interoperability problems or unexpected circumstances often prevent such an ideal situation to occur. In this Section, we report our attempt to reproduce a typical breakdown in a multihop ad-hoc network.

The setup is shown in Fig. 3 and a zoom on the fixed nodes is depicted in Fig. 4. A video captured during the experiment can be downloaded at [3]. The machine named $UMLx$ received and IP adresse of 192.168.0.x.

Obviously, this setup puts the system at loads and AODV messages are printed on the nodes console as soon as other nodes are coming in their reachability zone (as depicted by the yellow circles in Fig. 3). After letting the system run for a while, it was then found that a ping from $UML6$ to $UML3$ would only receive a single reply while a ping from $UML3$ to $UML6$ would receive no reply at all.

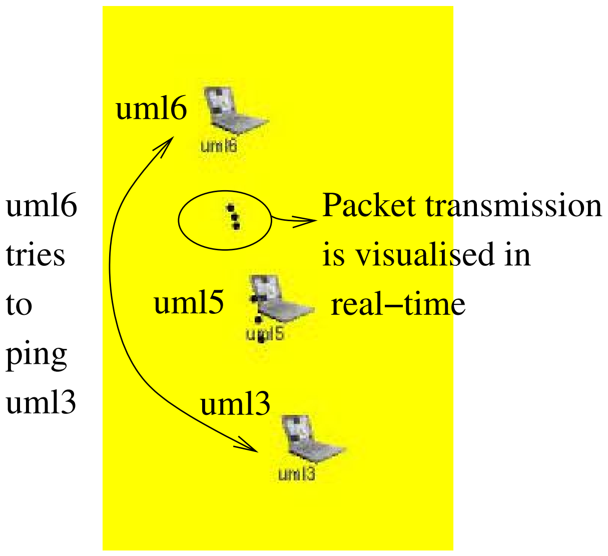


Figure 4: A zoom on the fixed nodes

A first tool at our disposal to understand what is happening is the use of the *tcpdump* packet capture program and the *ethereal* analyser. The trace acquired at $UML5$ is shown in Fig. 5. It can be seen in lines 35-38 that the ping request is arriving at $UML5$ and forwarded toward $UML3$ as is the first ping reply in the reversed direction. Lines 45-47 show the AODV hello messages with a lifetime of 3 seconds. The next trace, acquired at $UML03$ is shown in Fig. 6 with the Ethernet addresses where one can see the same sequence of two ping requests with one ping reply. However, the destination Ethernet address of the ping reply now comes as a surprise as it corresponds to the hardware address of $UML6$ showing that $UML3$ tries to bypass $UML5$. A look at Fig. 7 showing the AODV routing table of $UML6$ and $UML3$ reveals that $UML6$ is present in the routing table of $UML3$ while the reverse is not true. Indeed, it can be seen on the last line of Fig. 6 that $UML3$ can receive the hello messages from $UML6$.

A quick check revealed that the transmission power of $UML3$ had been modified at the beginning of the simulation and that its transmission range was therefore smaller than the transmission range of $UML6$. The first ping request was creating a reverse route on its way toward $UML3$ and the first reply was therefore reaching $UML6$. After the first hello message from $UML6$, the routing table in $UML3$ was modified and the ping replies were sent directly to $UML6$ which could not receive them because of the limited transmission power of $UML3$.

This practical example that actually occurred in the simulator is in fact widespread in practice and had been reported many times (for instance in [8]). It occurs notably when an AODV router is setup with a high power transmission while people trying to connect to it use a regular laptop card with smaller power transmission. The realistic physical transmission model used in the simulator allowed for recovering a typical situation often encountered in practice while the use of UML allowed for using conventional tools such as *tcpdump* for the debugging. This latter point might be of great practical interest for evaluating wireless oriented distribution before deploying them in the field.

4 A teaching tool

Although the results presented in the previous section did require an ergonomic simulation environment and a realistic enough physical layer emulation, the same results could have been obtained without the hassle of emulating in software the behaviour of a real hardware card. It would have been sufficient to code a UML specific driver just like the already existing tun/tap Ethernet driver. This would require, however, the complete rewriting of the userland interface, including the wireless extension. In this section, we present some results that specifically exploit the virtualisation of the wireless driver.

Indeed, with the help of the emulated hardware card, it is very easy to obtain detailed information about the internal mechanisms involved in the operation of the card. For many programmers, the interactions between the driver and the device remain a *terra incognita* that we may easily explore with our emulator. Fig. 8 and Fig. 9 show two different traces that can easily be obtained from the emulator software log file. The trace of the Fig. 8 can be read as follows :

1-3 The driver sets the parameter for a future command to $0x93c$ and send the command CMDCODE_ALLOC which interprets the given parameter as the size of the segment to allocate

4-5 Internally, the driver sets the event register bit corresponding to the memory allocation event and writes the ID of the allocated frame at the expected memory location

7-9 The events are acknowledged by the driver

This initialisation phase allocates memory regions referred to by their frame ID to be used later for packet transmission and reception. This is shown in Fig. 9 which display a trace corresponding to the transmission of a data frame. The *netbus_send* com-

```
35 8.502093    192.168.0.6          192.168.0.3          ICMP     Echo (ping) request
36 8.502277    192.168.0.6          192.168.0.3          ICMP     Echo (ping) request
37 8.537036    192.168.0.3          192.168.0.6          ICMP     Echo (ping) reply
38 8.537144    192.168.0.3          192.168.0.6          ICMP     Echo (ping) reply
45 9.581531    192.168.0.3          255.255.255.255      AODV
                RREP D: 192.168.0.3 O: 192.168.0.3 Hcnt=0 DSN=1 Lifetime=3000
46 9.999939    192.168.0.5          255.255.255.255      AODV
                RREP D: 192.168.0.5 O: 192.168.0.5 Hcnt=0 DSN=1 Lifetime=3000
47 10.336736   192.168.0.6          255.255.255.255      AODV
                RREP D: 192.168.0.6 O: 192.168.0.6 Hcnt=0 DSN=2 Lifetime=3000
60 12.386353   192.168.0.6          192.168.0.3          ICMP     Echo (ping) request
61 12.386556   192.168.0.6          192.168.0.3          ICMP     Echo (ping) request
62 12.418465   192.168.0.3          192.168.0.6          ICMP     Echo (ping) reply
```

Figure 5: packet capture on $UML5$ (from ethereal)

```
11:44:01.294387 0:c0:9f:16:27:4 0:c0:9f:16:10:3 0800
                98: 192.168.0.6 > 192.168.0.3: icmp: echo request (DF)
11:44:01.314788 0:c0:9f:16:10:3 0:c0:9f:16:6d:2 0800
                98: 192.168.0.6 > 192.168.0.3: icmp: echo request (DF)
11:44:01.314936 0:c0:9f:16:6d:2 0:c0:9f:16:27:4 0800
                98: 192.168.0.3 > 192.168.0.6: icmp: echo reply
11:44:01.326203 0:c0:9f:16:6d:2 0:c0:9f:16:27:4 0806 42:
                arp who-has 192.168.0.6 tell 192.168.0.3
11:44:01.676545 0:c0:9f:16:27:4 ff:ff:ff:ff:ff:ff 0800
                62: 192.168.0.6.654 > 255.255.255.255.654:  udp 20 (DF) [ttl 1]
```

Figure 6: packet capture on $UML3$ with the Ethernet addresses (from tcpdump)

```
Route Table at uml6
--------------------------------------------------------------------------------
        IP        |       Seq    |   Hop Count  |     Next Hop
--------------------------------------------------------------------------------
  192.168.0.2               1            1        192.168.0.2        Valid  s
ec/msec: 2/827 0
  192.168.0.5               1            1        192.168.0.5        Valid  s
ec/msec: 2/591 0
  192.168.0.6               1            0        192.168.0.6        Valid  s
ec/msec: 172874790/837 1
--------------------------------------------------------------------------------

Route Table at uml3
--------------------------------------------------------------------------------
        IP        |       Seq    |   Hop Count  |     Next Hop
--------------------------------------------------------------------------------
  192.168.0.7               1            1        192.168.0.7        Valid  E
xpired!
  192.168.0.6               1            1        192.168.0.6        Valid  s
ec/msec: 2/227 0
  192.168.0.5               1            1        192.168.0.5        Valid  s
ec/msec: 2/904 0
  192.168.0.3               1            0        192.168.0.3        Valid  s
ec/msec: 172874821/157 1
--------------------------------------------------------------------------------
```

Figure 7: AODV routing table at $UML6$ and $UML3$

```
(1) writew(0x93c,HFA384X_PARAM0_OFF)
(2) writew(0xa,HFA384X_CMD_OFF)
        (3) HFA384X_CMDCODE_ALLOC(0x93c,0x0)
(4) writew(HFA384X_EV_ALLOC,HFA384X_EV_STAT_OFF)
(5) writew(0x10,ALLOCFID_OFF)
        (6) evStat=0x18
(7) writew(0x10,HFA384X_EVACK_OFF)
(8) writew(0x8,HFA384X_EV_STAT_OFF)
(9) writew(0x8,HFA384X_EVACK_OFF)
```

Figure 8: Exchange between the hostap driver and the card during the card initialisation phase

```
(1)  writew(0x50,HFA384X_SELECT0_OFF)
(2)  writew(0x0,HFA384X_OFFSET0_OFF)
(3)  netbus_send a=HFA384X_DATA0_OFF len=60
(4)  writew(0x3c,HFA384X_OFFSET0_OFF)
(5)  netbus_send a=HFA384X_DATA0_OFF len=6
(6)  netbus_send a=HFA384X_DATA0_OFF len=1054
(7)  readw(0x30)
(8)  readw(0x38)
(9)  writew(0x460,HFA384X_OFFSET0_OFF)
(10) writew(0x50,HFA384X_PARAM0_OFF)
(11) writew(0x0,HFA384X_PARAM1_OFF)
(12) writew(0x10b,HFA384X_CMD_OFF)
        (13) HFA384X_CMDCODE_TRANSMIT(0x50,0x0)
(14) readw(0x38)
(15) Tx packet at 0x5000, len=1120
(16) readw(0x60)
(17) writew(0x8,0x60)
(18) writew(0x50,0x44)
(19) Interrupt evStat=0x8,inten=0xe09f
(20) writew(0x50,0x14)
        (21) evStat=0x18
(22) writew(0x10,HFA384X_EVACK_OFF)
(23) writew(0x8,0x60)
```

Figure 9: Exchange between the hostap driver and the card during the transmission of a data frame

mand corresponds to the extra feature mentioned previously that is used to transmit an entire buffer in one operation. Line (3) corresponds to the transmission of a frame descriptor and line (6) to the packet in itself. Once all the data have been transmitted, the transmit command itself is finally issued by the driver (line (10) to (13)) with the frame ID as parameter. Then, the card actually transmit the packet in the air (15) and the event status register bits are cleared. To our point of view, this kind of traces might be interesting to understand not only the static behaviour of a chipset as described in the datasheet but also the dynamics of the interactions between the driver and the card.

5 Related work

Several types of emulators have been proposed in the literature. In [10], Keshav et al. virtualised the networking stack of a FreeBSD kernel and allowed to run routing protocols and other networking applications in an emulated environment. Another similar approach is IMUNES proposed in [11]. IMUNES allows a FreeBSD kernel to maintain several networking stacks that are used to support different applications. However, those two solutions do not support wireless interfaces. As mentioned in the introduction, VNUML can be used to simulate wireless adhoc networks. However, this approach is less flexible as it requires an explicit description of the topology. Furthermore, communication links are bidirectional and symmetrical. In contrast, our physical model allows for a more realistic description of the transmission medium and the topology can be modified in real time by a simple “click-and-drag” operation.

Some debugging techniques related to Section 4 are described in [9], chap. 4 and include the use of a kernel debugger or of the Linux Trace Toolkit. Their also exists some I/O analysers that could be used to obtain some traces similar to those shown in Section 4. All these techniques could off course be used in conjunction with the tool presented in this paper. In particular, GDB can be used directly with UML and with the inserted modules. As the software card is “virtually plugged” into the kernel by inserting a module, the UML side of the card may easily be debugged and its interactions with the kernel may be studied with conventional tools.

Other emulation based solutions exist such as for instance *vmware*, *qemu* or *bochs*. While *vmware* does not allow for the addition of new emulated hardware components, *qemu* and *bochs*, which are free software could be used to accomplish the tasks described in this paper. For instance *bochs* supports an NE2000 compatible network card and a wireless interface could be added just as well. However, as stated in the bochs FAQ, bochs emulates every x86 instructions and all the devices in a PC system, it does not reach high emulation speeds.

6 Conclusion and future work

A virtual bus was proposed for User Mode Linux and used to insert the Linux *hostap* driver with nearly no modifications. By connecting multiple UML machines through a physical layer emulator and by providing a GUI for network visualisation, it was shown that this system could represent an interesting approach for wireless network emulation. The utility of such a method has been illustrated with two practical examples : the testing of an implementation of the AODV routing protocol in a highly realistic environment and the study of the interactions between the driver and the card it drives. Future work will now be focused on

- a complete implementation of the UML PCI interface to allow native drivers to run completely unmodified in UML.
- an implementation of the software card as a Linux kernel thread instead of a separate process. This would greatly improve the performance as it would no longer require any blocking operations for the synchronisation of the card and the kernel.
- improving the readability of the debugging facilities of the emulator. In particular, the *writew* operations initiated by the driver and the subsequent *writew* commands executed by the emulator itself should be reported separately. A GUI could be used to monitor the different status registers in real time.

References

[1] J. Dike. User mode linux. In *5th Annual Linux Showcase & conf.*, Oakland CA, 2001.

[2] D. Fernandez, Tomas de Miguel, and F. Galan. Study and emulation of ipv6 internet-exchange-based addressing models. *IEEE Communication Magazine*, pages 105–112, January 2004.

[3] V. Guffens. A wifi layer for user mode linux. http://www.auto.ucl.ac.be/uml-wifi/.

[4] V. Guffens, G. Bastin, and O. Bonaventure. An emulation infrastructure for multi-hop wireless communication networks. *Internal report*, 2004.

[5] Luke Klein-Berndt. Kernel aodv,national institue of standards and technology , http://w3.antd.nist.gov/wctg/aodv_kernel/.

[6] Jouni Malinen. Host ap driver for intersil prism2/2.5/3,http://hostap.epitest.fi/.

[7] User-mode-linux testing guide - openswan wiki. http://wiki.openswan.org/index.php/UMLTesting.

[8] Réseau citoyen. http://www.reseaucitoyen.be/.

[9] A. Rubini and J. Corbet. *Linux Device Drivers, 2^{nd} Ed.* O'Reilly, 2001.

[10] X.W.Huang, R. Sharma, and S. Keshav. The entrapid protocol development environment. In *Proc. of Infocom*, March 1999.

[11] Marko Zec. Implementing a clonable network stack in the freebsd kernel,. In *Proceedings of the 2003. USENIX Annual Technical Conference, San Antonio, Texas, June 2003.*

QEMU, a Fast and Portable Dynamic Translator

Fabrice Bellard

Abstract

We present the internals of QEMU, a fast machine emulator using an original portable dynamic translator. It emulates several CPUs (x86, PowerPC, ARM and Sparc) on several hosts (x86, PowerPC, ARM, Sparc, Alpha and MIPS). QEMU supports full system emulation in which a complete and unmodified operating system is run in a virtual machine and Linux user mode emulation where a Linux process compiled for one target CPU can be run on another CPU.

1 Introduction

QEMU is a machine emulator: it can run an unmodified *target* operating system (such as Windows or Linux) and all its applications in a virtual machine. QEMU itself runs on several host operating systems such as Linux, Windows and Mac OS X. The host and target CPUs can be different.

The primary usage of QEMU is to run one operating system on another, such as Windows on Linux or Linux on Windows. Another usage is debugging because the virtual machine can be easily stopped, and its state can be inspected, saved and restored. Moreover, specific embedded devices can be simulated by adding new machine descriptions and new emulated devices.

QEMU also integrates a Linux specific user mode emulator. It is a subset of the machine emulator which runs Linux processes for one target CPU on another CPU. It is mainly used to test the result of cross compilers or to test the CPU emulator without having to start a complete virtual machine.

QEMU is made of several subsystems:

- CPU emulator (currently x86[1], PowerPC, ARM and Sparc)
- Emulated devices (e.g. VGA display, 16450 serial port, PS/2 mouse and keyboard, IDE hard disk, NE2000 network card, ...)
- Generic devices (e.g. block devices, character devices, network devices) used to connect the emulated devices to the corresponding host devices
- Machine descriptions (e.g. PC, PowerMac, Sun4m) instantiating the emulated devices
- Debugger
- User interface

This article examines the implementation of the dynamic translator used by QEMU. The dynamic translator performs a runtime conversion of the target CPU instructions into the host instruction set. The resulting binary code is stored in a translation cache so that it can be reused. The advantage compared to an interpreter is that the target instructions are fetched and decoded only once.

Usually dynamic translators are difficult to port from one host to another because the whole code generator must be rewritten. It represents about the same amount of work as adding a new target to a C compiler. QEMU is much simpler because it just concatenates pieces of machine code generated *off line* by the GNU C Compiler [5].

A CPU emulator also faces other more classical but difficult [2] problems:

- Management of the translated code cache
- Register allocation
- Condition code optimizations
- Direct block chaining
- Memory management
- Self-modifying code support

- Exception support
- Hardware interrupts
- User mode emulation

2 Portable dynamic translation

2.1 Description

The first step is to split each target CPU instruction into fewer simpler instructions called *micro operations*. Each micro operation is implemented by a small piece of C code. This small C source code is compiled by GCC to an object file. The micro operations are chosen so that their number is much smaller (typically a few hundreds) than all the combinations of instructions and operands of the target CPU. The translation from target CPU instructions to micro operations is done entirely with hand coded code. The source code is optimized for readability and compactness because the speed of this stage is less critical than in an interpreter.

A compile time tool called `dyngen` uses the object file containing the micro operations as input to generate a dynamic code generator. This dynamic code generator is invoked at runtime to generate a complete host function which concatenates several micro operations.

The process is similar to [1], but more work is done at compile time to get better performance. In particular, a key idea is that in QEMU constant parameters can be given to micro operations. For that purpose, dummy code relocations are generated with GCC for each constant parameter. This enables the `dyngen` tool to locate the relocations and generate the appropriate C code to resolve them when building the dynamic code. Relocations are also supported to enable references to static data and to other functions in the micro operations.

2.2 Example

Consider the case where we must translate the following PowerPC instruction to x86 code:

```
addi r1,r1,-16   # r1 = r1 - 16
```

The following micro operations are generated by the PowerPC code translator:

```
movl_T0_r1          # T0 = r1
addl_T0_im -16      # T0 = T0 - 16
movl_r1_T0          # r1 = T0
```

The number of micro operations is minimized without impacting the quality of the generated code much. For example, instead of generating every possible move between every 32 PowerPC registers, we just generate moves to and from a few temporary registers. These registers `T0`, `T1`, `T2` are typically stored in host registers by using the GCC static register variable extension.

The micro operation `movl_T0_r1` is typically coded as:

```
void op_movl_T0_r1(void)
{
    T0 = env->regs[1];
}
```

`env` is a structure containing the target CPU state. The 32 PowerPC registers are stored in the array `env->regs[32]`.

`addl_T0_im` is more interesting because it uses a *constant parameter* whose value is determined at runtime:

```
extern int __op_param1;
void op_addl_T0_im(void)
{
    T0 = T0 + ((long)(&__op_param1));
}
```

The code generator generated by `dyngen` takes a micro operation stream pointed by `opc_ptr` and outputs the host code at position `gen_code_ptr`. Micro operation parameters are pointed by `opparam_ptr`:

```
[...]
for(;;) {
  switch(*opc_ptr++) {
  [...]
  case INDEX_op_movl_T0_r1:
  {
    extern void op_movl_T0_r1();
    memcpy(gen_code_ptr,
      (char *)&op_movl_T0_r1+0,
      3);
    gen_code_ptr += 3;
    break;
  }
  case INDEX_op_addl_T0_im:
  {
    long param1;
    extern void op_addl_T0_im();
    memcpy(gen_code_ptr,
      (char *)&op_addl_T0_im+0,
      6);
    param1 = *opparam_ptr++;
    *(uint32_t *)(gen_code_ptr + 2) =
      param1;
    gen_code_ptr += 6;
    break;
  }
```

```
    [...]
    }
}
[...]
}
```

For most micro operations such as `movl_T0_r1`, the host code generated by GCC is just copied. When constant parameters are used, `dyngen` uses the fact that relocations to `__op_param1` are generated by GCC to patch the generated code with the runtime parameter (here it is called `param1`).

When the code generator is run, the following host code is output:

```
# movl_T0_r1
# ebx = env->regs[1]
mov    0x4(%ebp),%ebx
# addl_T0_im -16
# ebx = ebx - 16
add    $0xfffffff0,%ebx
# movl_r1_T0
# env->regs[1] = ebx
mov    %ebx,0x4(%ebp)
```

On x86, `T0` is mapped to the `ebx` register and the CPU state context to the `ebp` register.

2.3 Dyngen implementation

The `dyngen` tool is the key of the QEMU translation process. The following tasks are carried out when running it on an object file containing micro operations:

- The object file is parsed to get its symbol table, its relocations entries and its code section. This pass depends on the host object file format (`dyngen` supports ELF (Linux), PE-COFF (Windows) and MACH-O (Mac OS X)).

- The micro operations are located in the code section using the symbol table. A host specific method is executed to get the start and the end of the copied code. Typically, the function prologue and epilogue are skipped.

- The relocations of each micro operations are examined to get the number of constant parameters. The constant parameter relocations are detected by the fact they use the specific symbol name `__op_paramN`.

- A memory copy in C is generated to copy the micro operation code. The relocations of the code of each micro operation are used to patch the copied code so that it is properly relocated. The relocation patches are host specific.

- For some hosts such as ARM, constants must be stored near the generated code because they are accessed with PC relative loads with a small displacement. A host specific pass is done to relocate these constants in the generated code.

When compiling the micro operation code, a set of GCC flags is used to manipulate the generation of function prologue and epilogue code into a form that is easy to parse. A dummy assembly macro forces GCC to always terminate the function corresponding to each micro operation with a single return instruction. Code concatenation would not work if several return instructions were generated in a single micro operation.

3 Implementation details

3.1 Translated Blocks and Translation Cache

When QEMU first encounters a piece of target code, it translates it to host code up to the next jump or instruction modifying the *static CPU state* in a way that cannot be deduced at translation time. We call these basic blocks Translated Blocks (TBs).

A 16 MByte cache holds the most recently used TBs. For simplicity, it is completely flushed when it is full.

The static CPU state is defined as the part of the CPU state that is considered as known at translation time when entering the TB. For example, the program counter (PC) is known at translation time on all targets. On x86, the static CPU state includes more data to be able to generate better code. It is important for example to know if the CPU is in protected or real mode, in user or kernel mode, or if the default operand size is 16 or 32 bits.

3.2 Register allocation

QEMU uses a fixed register allocation. This means that each target CPU register is mapped to a fixed host register or memory address. On most hosts, we simply map all the target registers to memory and only store a few temporary variables in host registers. The allocation of the temporary variables is hard coded in each target CPU description. The advantage of this method is simplicity and portability.

The future versions of QEMU will use a dynamic temporary register allocator to eliminate some unnecessary moves in the case where the target registers are directly stored in host registers.

3.3 Condition code optimizations

Good CPU condition code emulation (`eflags` register on x86) is a critical point to get good performances.

QEMU uses lazy condition code evaluation: instead of computing the condition codes after each x86 instruction, it just stores one operand (called CC_SRC), the result (called CC_DST) and the type of operation (called CC_OP). For a 32 bit addition such as $R = A + B$, we have:

```
CC_SRC=A
CC_DST=R
CC_OP=CC_OP_ADDL
```

Knowing that we had a 32 bit addition from the constant stored in CC_OP, we can recover A, B and R from CC_SRC and CC_DST. Then all the corresponding condition codes such as zero result (ZF), non-positive result (SF), carry (CF) or overflow (OF) can be recovered if they are needed by the next instructions.

The condition code evaluation is further optimized at translation time by using the fact that the code of a complete TB is generated at a time. A backward pass is done on the generated code to see if CC_OP, CC_SRC or CC_DST are not used by the following code. At the end of TB we consider that these variables are used. Then we delete the assignments whose value is not used in the following code.

3.4 Direct block chaining

After each TB is executed, QEMU uses the simulated Program Counter (PC) and the other information of the static CPU state to find the next TB using a hash table. If the next TB has not been already translated, then a new translation is launched. Otherwise, a jump to the next TB is done.

In order to accelerate the most common case where the new simulated PC is known (for example after a conditional jump), QEMU can patch a TB so that it jumps directly to the next one.

The most portable code uses an indirect jump. On some hosts (such as x86 or PowerPC), a branch instruction is directly patched so that the block chaining has no overhead.

3.5 Memory management

For system emulation, QEMU uses the `mmap()` system call to emulate the target MMU. It works as long as the emulated OS does not use an area reserved by the host OS.[2]

In order to be able to launch any OS, QEMU also supports a *software MMU*. In that mode, the MMU virtual to physical address translation is done at every memory access. QEMU uses an address translation cache to speed up the translation.

To avoid flushing the translated code each time the MMU mappings change, QEMU uses a physically indexed translation cache. It means that each TB is indexed with its physical address.

When MMU mappings change, the chaining of the TBs is reset (i.e. a TB can no longer jump directly to another one) because the physical address of the jump targets may change.

3.6 Self-modifying code and translated code invalidation

On most CPUs, self-modifying code is easy to handle because a specific code cache invalidation instruction is executed to signal that code has been modified. It suffices to invalidate the corresponding translated code.

However on CPUs such as the x86, where no instruction cache invalidation is signaled by the application when code is modified, self-modifying code is a special challenge .[3]

When translated code is generated for a TB, the corresponding host page is write protected if it is not already read-only. If a write access is made to the page, then QEMU invalidates all the translated code in it and re-enables write accesses to it.

Correct translated code invalidation is done efficiently by maintaining a linked list of every translated block contained in a given page. Other linked lists are also maintained to undo direct block chaining.

When using a software MMU, the code invalidation is more efficient: if a given code page is invalidated too often because of write accesses, then a bitmap representing all the code inside the page is built. Every store into that page checks the bitmap to see if the code really needs to be invalidated. It avoids invalidating the code when only data is modified in the page.

3.7 Exception support

`longjmp()` is used to jump to the exception handling code when an exception such as division by zero is encountered. When not using the software MMU, host signal handlers are used to catch the invalid memory accesses.

QEMU supports precise exceptions in the sense that it is always able to retrieve the exact target CPU state at the time the exception occurred.[4] Nothing has to be done for most of the target CPU state because it is explicitly stored and modified by the translated code. The target CPU state S which is not explicitly stored (for example the current Program Counter) is retrieved by re-translating the TB where the exception occurred in a mode where S is recorded before each translated target instruction. The host program counter where the exception was raised is used to find the corresponding target instruction and the state S.

3.8 Hardware interrupts

In order to be faster, QEMU does not check at every TB if an hardware interrupt is pending. Instead, the user must asynchronously call a specific function to tell that an interrupt is pending. This function resets the chaining of the currently executing TB. It ensures that the execution will return soon in the main loop of the CPU emulator. Then the main loop tests if an interrupt is pending and handles it.

3.9 User mode emulation

QEMU supports user mode emulation in order to run a Linux process compiled for one target CPU on another CPU.

At the CPU level, user mode emulation is just a subset of the full system emulation. No MMU simulation is done because QEMU supposes the user memory mappings are handled by the host OS. QEMU includes a generic Linux system call converter to handle endianness issues and 32/64 bit conversions. Because QEMU supports exceptions, it emulates the target signals exactly. Each target thread is run in one host thread[5].

4 Porting work

In order to port QEMU to a new host CPU, the following must be done:

- `dyngen` must be ported (see section 2.2).
- The temporary variables used by the micro operations may be mapped to host specific registers in order to optimize performance.
- Most host CPUs need specific instructions in order to maintain coherency between the instruction cache and the memory.
- If direct block chaining is implemented with patched branch instructions, some specific assembly macros must be provided.

The overall porting complexity of QEMU is estimated to be the same as the one of a dynamic linker.

5 Performance

In order to measure the overhead due to emulation, we compared the performance of the BYTEmark benchmark for Linux [7] on a x86 host in native mode, and then under the x86 target user mode emulation.

User mode QEMU (version 0.4.2) was measured to be about 4 times slower than native code on integer code. On floating point code, it is 10 times slower. This can be understood as a result of the lack of the x86 FPU stack pointer in the static CPU state. In full system emulation, the cost of the software MMU induces a slowdown of a factor of 2.

In full system emulation, QEMU is approximately 30 times faster than Bochs [4].

User mode QEMU is 1.2 times faster than `valgrind --skin=none` version 1.9.6 [6], a hand coded x86 to x86 dynamic translator normally used to debug programs. The `--skin=none` option ensures that Valgrind does not generate debug code.

6 Conclusion and Future Work

QEMU has reached the point where it is usable in everyday work, in particular for the emulation of commercial x86 OSes such as Windows. The PowerPC target is close to launch Mac OS X and the Sparc one begins to launch Linux. No other dynamic translator to date has supported so many targets on so many hosts, mainly because the porting complexity was underestimated. The QEMU approach seems a good compromise between performance and complexity.

The following points still need to be addressed in the future:

- Porting: QEMU is well supported on PowerPC and x86 hosts. The other ports on Sparc, Alpha, ARM and MIPS need to be polished. QEMU also depends very much on the exact GCC version used to compile the micro operations definitions.
- Full system emulation: ARM and MIPS targets need to be added.
- Performance: the software MMU performance can be increased. Some critical micro operations can also be hand coded in assembler without much modifications in the current translation framework. The CPU main loop can also be hand coded in assembler.
- Virtualization: when the host and target are the same, it is possible to run most of the code as is. The simplest implementation is to emulate the target kernel code as usual but to run the target user code as is.
- Debugging: cache simulation and cycle counters could be added to make a debugger as in SIMICS [3].

7 Availability

QEMU is available at

`http://bellard.org/qemu`

References

[1] Ian Piumarta, Fabio Riccardi, Optimizing direct threaded code by selective inlining, Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[2] Mark Probst, Fast Machine-Adaptable Dynamic binary Translation, Workshop on Binary Translation 2001.

[3] Peter S. Magnusson et al., SimICS/sun4m: A Virtual Workstation, Usenix Annual Technical Conference, June 15-18, 1998.

[4] Kevin Lawton et al., the Bochs IA-32 Emulator Project, `http://bochs.sourceforge.net`.

[5] The Free Software Foundation, the GNU Compiler Collection, `http://gcc.gnu.org`.

[6] Julian Seward et al., Valgrind, an open-source memory debugger for x86-GNU/Linux, `http://valgrind.kde.org/`.

[7] The BYTEmark benchmark program, BYTE Magazine, Linux version available at `http://www.tux.org/~mayer/linux/bmark.html`.

Notes

[1] *x86* CPUs refer to processors compatible with the Intel 80386 processor.

[2] This mode is now deprecated because it needs a patched target OS and because the target OS can access to the host QEMU address space.

[3] For simplicity, QEMU will in fact implement this behavior of ignoring the code cache invalidation instructions for all supported CPUs.

[4] In the x86 case, the virtual CPU cannot retrieve the exact `eflags` register because in some cases it is not computed because of condition code optimizations. This is not a major concern because the emulated code can still be restarted at the same point in any cases.

[5] At the time of writing, QEMU's support for threading is considered to be immature due to locking issues within its CPU core emulation.

USB/IP - a Peripheral Bus Extension for Device Sharing over IP Network

Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, 630-0192, Japan
Email: {taka-hir, eiji-ka}@is.naist.jp, fujikawa@itc.naist.jp, suna@wide.ad.jp

Abstract

As personal computing becomes more popular and affordable, the availability of peripheral devices is also increasing rapidly. However, these peripheral devices can usually only be connected to a single machine at time. The ability to share peripheral devices between computers without any modification of existing computing environments is, consequently, a highly desirable goal, as it improves the efficiency and usability of such devices. Existing device sharing technologies in the pervasive computing area are not sufficient for peripheral devices designed for personal computers, because these technologies do not provide the degree of network-transparency necessary for both applications and device drivers.

In this paper, we propose USB/IP as a *peripheral bus extension* over an Internet Protocol (IP) network. This novel device sharing approach is based on the sophisticated peripheral interfaces that are supported in most modern operating systems. Using a *virtual peripheral bus driver*, users can share a diverse range of devices over networks without any modification in existing operating systems and applications. Our experiments show that USB/IP has sufficient I/O performance for many USB devices, including isochronous ones. We also describe performance optimization criteria that can be used to achieve further performance improvements.

1 Introduction

Recent innovations in computing technology have enabled people to establish their own computing environments that comprise multiple personal computers connected together via a local area network. In such an environment, the ability to access peripheral devices attached to one computer, seamlessly and on-demand from another computer, is a highly desirable attribute. For example, a user who brings back a mobile computer to the office may want to make the backup of working files directly onto the DVD-R drive of a shared computer, rather than directly use the shared computer or move the DVD-R drive. The user may also wish to work on the mobile computer using an ergonomic keyboard and a mouse which is already attached to another a desktop computer, but without connecting to a KVM switch. In the context of resource management, the key technology for these scenarios is network-transparent device sharing by which one computer can interact seamlessly with another computer's devices in addition to directly-attached devices.

Many device sharing technologies have been proposed in the pervasive computing area, to aggregate accesses to network-attached devices and improve their usability. These technologies address dynamic discovery, on-demand selection, and automatic interaction among devices. However, little work has been done to ensure network transparency for existing device access interfaces, so that existing applications can access remote shared devices without any modification.

In this paper, we propose USB/IP as a *peripheral bus extension* over an Internet Protocol (IP) network. Our philosophy is to allow a computer to extend its local peripheral buses over an IP network to another computer. The main component of this extension is a *virtual peripheral bus driver* which provides a virtual extension of the standard peripheral bus driver. The driver resides at the lowest layer in the operating system so that most applications and device drivers can access remote shared devices through existing interfaces. We believe that this approach particularly suits recently emerging sophisticated peripheral interfaces and broadband networks.

Our device sharing approach presents several advantages over the conventional approaches. First, computers can access the full functionality of a remote shared device. The control granularity of the shared device is the same as for a directly-attached device. In our system, all the low-level control commands for a device are encapsulated into IP packets and then transmitted. Second, computers can access a shared device using standard operating systems and applications. By installing only a few additional device drivers, it is possible to control a remote device as if it was directly attached to the local peripheral bus. Third, various computers with different operating systems can share their devices with each other, because the low-level device control protocols do not depend on any operating system specific information. Last, most devices can be shared in this way, because a virtual peripheral bus driver is independent of other device drivers, and supports all the devices on its peripheral interface.

In the remainder of this paper, we expand on our vision of the peripheral bus extension. Section 2 explains the research motivation and the advantages of USB/IP from the viewpoint of the device sharing architecture. Section 3 describes the design and implementation details of USB/IP. Section 4 shows the evaluation of USB/IP and clarifies its characteristics. Section 5 provides some additional discussion and then Section 6 examines related work. We conclude our study in Section 7. The availability of USB/IP is noted in Section 8.

2 Peripheral Bus Extension

This section describes the importance of USB/IP as a new device sharing approach for peripheral bus extension. We explain what motivates the new sharing approach in contrast to conventional techniques. Finally, we explain the concept of the peripheral bus extension in the context of the device driver models of operating systems.

2.1 Motivation

The responsibility of an operating system is to manage various resources on a computer or network and to provide applications with access to these resources through generalized interfaces. A peripheral device attached to a computer is managed as one of these resources, and so is accessed by applications through a common interface which provide an abstract representation of hardware device functions (e.g., read and write). Most conventional device sharing models only allow applications to use remote devices through these abstract functions; computers do not share any more fine-grained operations than the abstract functions. Since shared devices in these models are controlled with only a few high-level operations, it is possible to manipulate the remote shared devices under narrow bandwidth restrictions, or where large network delays are present. In addition, concurrent access to a remote device can be controlled using a locking mechanism that is applied to the abstract data unit of the resource.

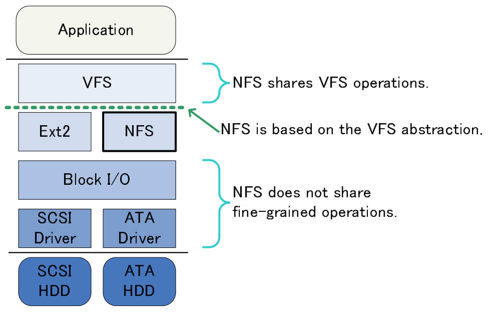

Application
VFS
NFS shares VFS operations.
Ext2
NFS
NFS is based on the VFS abstraction.
Block I/O
NFS does not share fine-grained operations.
SCSI Driver
ATA Driver
SCSI HDD
ATA HDD

Figure 1: Relation between Device Abstraction and Device Sharing (an example of storage device sharing by NFS)

Taking a storage device as an example, the most popular unit of data manipulation is a *file*. Files are organized into file systems and applications interact with them through system calls. When a hard disk is shared by NFS [15], computers control the remote shared storage device with the remote procedure calls similar to the system calls provided for direct file operations. Though the computers cannot deal with the remote storage device directly, this mechanism works fine as long as it is limited to storing and retrieving data.

While the conventional device sharing models were acceptable approaches in the past, they do not work well for sharing many of the new peripheral devices. The device sharing system should be able to maximize the use of new-featured remote devices, and also provide applications with seamless and on-demand access to both local and remote devices. However, the conventional device sharing models do not fill these requirements for the following reasons:

First, only abstract operations are sharable, so the more fine-grained and device-specific operations are not supported (Figure 1). This precludes the use of remote shared devices in the same way as directly-attached devices.

Since NFS is implemented using the virtual file system (VFS) layer in the UNIX operating system, it is not able to share either the common APIs for block devices nor the native I/O operations for ATA or SCSI disks, both of which are lower-level functions than the VFS. For example, the NFS protocol does not define methods to format a remote storage device, or to eject a remote removable media device.

Second, while the original functions of both a remote device and a locally-attached device are identical, the control interfaces for both devices are often different. Most control procedures for locally-attached devices are implemented in device drivers. However, sharing mechanisms to access remote devices are often implemented in the upper layer of the operating system, such as userland applications or libraries. This gap between both interfaces forces developers to implement a new dedicated application, or to modify existing applications to support this functionality.

For instance, VNC [12] provides framebuffer sharing with remote computers by transmitting the screen image data continuously. However, a VNC client implemented as a userland application does not provide the same access interface as the local physical framebuffer. The applications or drivers that directly handle physical framebuffers, such as console terminal drivers, cannot get the benefit of the framebuffer sharing through VNC.

Third, to achieve a high degree of interoperability is sometimes difficult for a device sharing system because of the complex differences between operating systems. Some device sharing applications (those that just extend the existing abstraction layer to forward device requests) usually only support the same operating system and cannot interoperate with other operating systems which do not have such an abstraction layer. Furthermore, the interoperability sometimes conflicts with the above first issue; the abstraction for bridging different operating systems usually disables some specific functions of shared devices.

RFS [13] provides transparent access to remote files by preserving most UNIX file system semantics, and allows all file types, including special devices and named pipes, to be shared. RFS allows access to remote devices by mounting the remote device files in the local file system tree. However, RFS is not even generic enough to connect between different versions of the UNIX operating system because of the diverse semantics of certain operations, such as `ioctl()` arguments.

Finally, peripheral devices with new features are constantly being developed. These features are often not compatible with the existing ones, and so cannot be shared without extensions to the device sharing system. In pervasive computing, the ability to adopt new technology as soon as it becomes available is a key goal. Providing remote device sharing for these new devices as soon as they are released requires a more general device sharing mechanism that can be applied to any device easily.

2.2 Peripheral Bus Extension Philosophy

In traditional operating systems, the device driver is responsible for communication with the physical device, and for providing an interface to allow userland applications to access the device functionality (Figure 2 left). This model forces conventional device sharing systems to evolve with the drawbacks described in Section 2.1.

Major advances in computer technology are now changing many of the technical issues that have compelled the conventional device sharing approach. Computer hardware is rapidly improving, and the wide availability of intelligent peripheral buses (e.g., USB and IEEE1394) is now commonplace. These peripheral buses have advanced features such as serialized I/O, plug-and-play, and universal connectivity. Serialized I/O, which supports devices with a wide range of I/O speeds, minimizes hardware and software implementation costs by reducing (or eliminating) the need for the legacy interfaces (e.g., PS/2, RS-232C, IEEE1284 and ISA). It also improves the I/O performance by employing an efficient bus arbitration mechanism for data transmission. Plug-and-play simplifies device connection by allowing dynamic attachment and automatic configuration of devices. Universal connectivity provides multiple transport modes, including isochronous transfer for multimedia devices, which improves the usability of the bus for a wide range of devices.

Device drivers are generally responsible for data transmission and communication with the attached device, while the operating system must provide the management framework for dynamic device configuration. Furthermore, a device driver is normally separated into a bus driver for manipulation of peripheral interfaces, and a per-device driver for control of devices on the peripheral interfaces. (Figure 2 center).

The USB/IP device sharing approach extends the peripheral bus over an IP network using a *virtual bus driver* (Figure 2 right). The virtual bus driver provides an interface to remote shared devices by encapsulating peripheral bus request commands in IP packets and transmitting them across the network. This approach has the advantage of being able to utilize the existing dynamic device management mechanism in the operating system, for shared devices.

Our approach resolves the drawbacks of the conventional approaches, and also has several advantages:

Full Functionality. All the functions supported by remote devices can be manipulated by the operating system. The shared operations are implemented in the lowest layer, so the abstraction at the bus driver layer conceals only the bus differences and does not affect the per-device operations. In the kernel structure, both the locally-attached devices and the remote devices are positioned in the same layer.

Network Transparency. In this paper, we define the network transparency as “shared devices on the network can be controlled by existing operating systems and applications without any modification”. The virtual bus driver can conceal the implementation details of network sharing mechanisms. Shared devices are controlled by existing device drivers through the virtual bus driver. Other components of the operating system (e.g., file system, block I/O and virtual memory) and applications do not notice any difference between the access interfaces of shared devices and locally-attached ones.

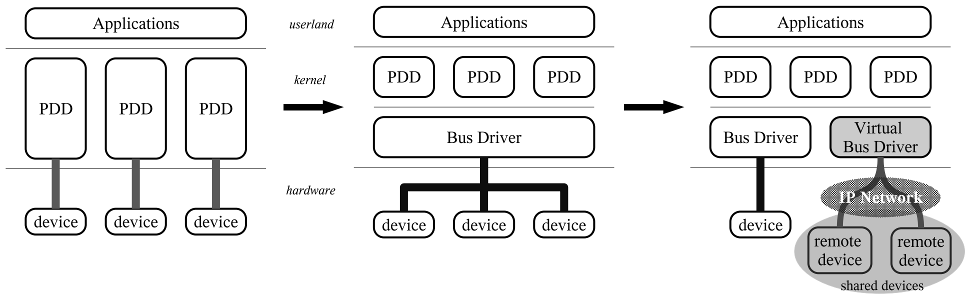


Figure 2: Evolution of Device Driver Framework
Device driver models are illustrated; conventional peripheral interfaces (left), sophisticated peripheral interfaces (center) and our proposed approach (right). PDD is the abbreviation of per-device driver.

Interoperability. The device sharing system based uses the low-level control protocols of the peripheral devices, which are the same as those of locally attached devices. These protocols, which are defined by several standards, are independent of the operating system implementation. By using these protocols for sharing, computers with different operating systems can easily share and control devices remotely.

Generality. Most devices can be shared in this way, because a virtual bus driver is independent of the per-device drivers, and supports all the devices on its peripheral interface. This allows the sharing of diverse devices over an IP network.

There are some issues that need to be considered when applying the USB/IP model in practice. First, the proposed approach is not able to provide concurrent access to a remote peripheral device, since the raw device functions are being shared. In contrast, conventional device sharing often allow concurrent access. However, our approach affects a lower layer of an operating system than the conventional approaches do; it is possible that both approaches are complementary to each other. Also, devices which support multiple simultaneous accesses, such as shared SCSI, can be shared by employing a network lock mechanism.

Second, the control protocols used by peripheral buses are not designed for transmission over IP networks. Using these protocols over an IP network raises some issues about network delay and jitters. However, high-speed network technologies, such as Gigabit Ethernet, are now able to provide bandwidths that are of the same order as those of modern peripheral interfaces. In addition, the rapid progress of networking technologies will alleviate many of these issues in the near future.

In the next section, we demonstrate the suitability of peripheral bus extension of sophisticated device interfaces by describing a prototype implementation and showing its validity through various experiments. We also discuss issues for the control of devices over IP networks that have arisen from these results.

3 USB/IP

In this section, we describe a practical example of the peripheral bus extension for the USB protocol, called USB/IP. We first discuss the USB device driver model and the granularity of operations, and then describe a strategy for IP encapsulation of the USB protocol.

3.1 USB Device Driver Model

USB (Universal Serial Bus) is one of the more sophisticated peripheral interfaces, providing serialized I/O, dynamic device configuration and universal connectivity. The USB 2.0 specification, announced in April 2000, specifies that a host computer can control various devices using 3 transfer speeds (1.5Mbps, 12.0Mbps, and 480Mbps) and 4 transfer types (Control, Bulk, Interrupt, and Isochronous).

Data transmission using the Isochronous and Interrupt transfer types is periodically scheduled. The Isochronous transfer type is used to transmit control data at a constant bit rate, which is useful for reading image data from a USB camera, or for writing sound data to a USB speaker. The shortest I/O transaction interval supported using Isochronous transfers, called a microframe, is 125us. The Interrupt transfer type negotiates the maximum delay allowed for a requested transaction. This is primarily used for USB mice and keyboards, which require a small amount of data to be sent periodically in a short interval.

The Control and Bulk transfer types are asynchronously scheduled into the bandwidth remaining after the periodic transfers have been scheduled. The Control transfer type is used primarily for enumeration and initialization of devices. In 480Mbps mode, 20% of the total bandwidth is reserved for Control transfer. The Bulk

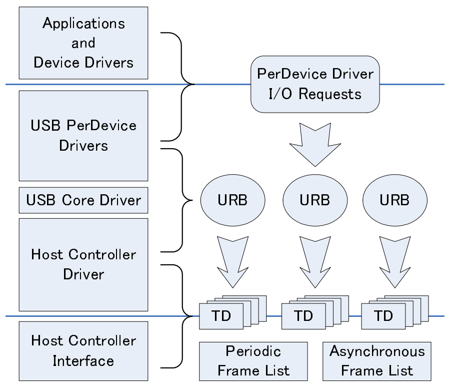


Figure 3: USB Device Driver Model

transfer type is used for the requests that have no temporal restrictions, such as storage device I/O. This is the fastest transfer mode when the bus is available.

The device driver model of USB is layered as illustrated in Figure 3. At the lower layer, the granularity of operations is fine-grained; the data size and temporal restriction of each operation are smaller than in the upper layer.

3.1.1 USB Per-Device Driver

USB Per-Device Drivers (PDDs) are responsible for controlling individual USB devices. When applications or other device drivers request I/O to a USB device, a USB PDD converts the I/O requests to a series of USB commands and then submits them to a USB Core Driver in the form of USB Request Blocks (URBs). A USB PDD uses only a device address, an endpoint address, an I/O buffer and some additional information required for each transfer type, to communicate with the device. USB PDDs do not need to interact with the hardware interfaces or registers of the host controllers, nor do they modify the IRQ tables.

3.1.2 USB Core Driver

A USB Core Driver is responsible for the dynamic configuration and management of USB devices. When a new USB device is attached to the bus, it is enumerated by the Core Driver, which requests device specific information from it and then loads the appropriate USB PDD.

A USB Core Driver also provides a set of common interfaces to the upper USB PDDs and the lower USB Host Controller Drivers. A USB PDD submits a USB request to the device via the Host Controller Driver. This request is in the form of a URB. Notification of the completed request is provided using a completion handler in the URB. This is an asynchronous process from the view point of the I/O model.

3.1.3 USB Host Controller Driver

A USB Host Controller Driver (HCD) receives URBs from a USB Core Driver and then divides them into smaller requests, known as Transfer Descriptors (TDs), which correspond to the USB microframes. TDs are scheduled depending on their transfer types and are linked to the appropriate frame lists in the HCD for delivery. TDs for Isochronous or Interrupt transfer types are linked to the Periodic Frame List and Bulk and Control transfer types are linked to the Asynchronous Frame List. The actual work of the I/O transaction is performed by the host controller chip.

3.2 IP Encapsulation Strategy

To implement the peripheral bus extension of USB, we have added a *Virtual Host Controller Interface* (VHCI) driver as a virtual bus driver (as described in Section 2.2). The VHCI driver is the equivalent of a USB HCD, and is responsible for processing enqueued URBs. A URB is converted into a USB/IP request block by the VHCI driver and sent to the remote machine. A *Stub* driver is also added as a new type of USB PDD. The Stub driver is responsible for decoding incoming USB/IP packets from remote machines, extracting the URBs, and then submitting them to the local USB devices.

Using this strategy, any interface differences between directly-attached USB devices and remote USB devices is completely hidden by the HCD layer. USB PDDs, other drivers, and applications can use remote USB devices in exactly the same way. Once a USB Core Driver enumerates and initializes the remote USB devices, unmodified USB PDDs and applications can access the devices as if they were locally attached.

An IP network typically has a significantly larger transfer delay and more jitter than the USB network. In addition, the native I/O granularity of USB is too small to effectively control USB devices over an IP network. The Isochronous transfer type needs to transmit 3KB of data in every microframe (125us), and the Bulk transfer type needs to transmit 6.5KB of data in a microframe. Therefore, to transfer USB commands over the IP network efficiently, USB/IP is designed to encapsulate a URB (not a TD) into IP packets. This technique minimizes these timing issues by concatenating a series of USB I/O transactions into a single URB. For example, using the Isochronous transfer type, by combining 80 I/O transactions that are executed every microframe into a single URB, the packet can be delayed 10ms, while still maintaining an I/O granularity of 125us. Similarly, using the Bulk transport type, a URB that has a 100KB buffer can be used to transfer 200 I/O transactions containing 512B of data each.

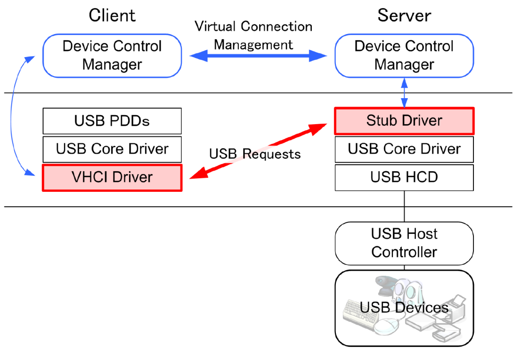


Figure 4: USB/IP Design

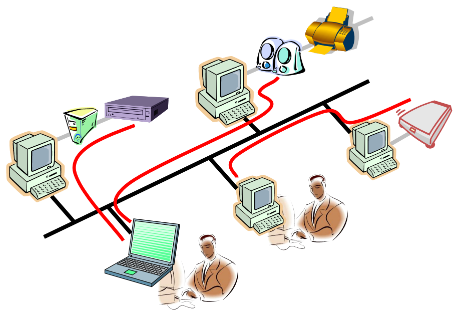


Figure 5: A USB/IP Application (Device Sharing for LAN)

```
% devconfig list                  list available remote devices

  3: IO-DATA DEVICE,INC. Optical Storage
   : IP:PORT       : 10.0.0.2:3000
   : local_state   : CONN_DONE
   : remote_state  : INUSE
   : remote_user   : 10.0.0.3
   : remote_module: USBIP

  2: Logitech M4848
   : IP:PORT       : 10.0.0.2:3000
   : local_state   : DISCONN_DONE
   : remote_state  : AVAIL
   : remote_user   : NOBODY
   : remote_module: USBIP

% devconfig up 3                  attach a remote DVD Drive
% ...
% ...                      mount/read/umount a DVD-ROM
% ...                                  play a DVD movie
% ...                        record data to a DVD-R media
% ...
% devconfig down 3                detach a remote DVD Drive
```

Figure 6: USB/IP Application Usage

These issues, while important for slower networks, are minimized by new network technologies, such as Gigabit Ethernet. The bandwidth of Gigabit Ethernet is significantly greater than the 480Mbps used by USB 2.0, so it is possible to apply the URB-based I/O model to control remote USB devices over such a network. We evaluate the use of URBs for USB/IP is more detail in Section 4.

3.3 Design and Implementation of USB/IP

The design of USB/IP is illustrated in Figure 4. A VHCI driver acts as a USB HCD in the client host, and a Stub driver acts as a USB PDD in the server host. The VHCI driver emulates the USB Root Hub's behavior, so when a remote USB device is connected to a client host over the IP network, the VHCI driver notifies the USB Core Driver of the port status change. The USB/IP driver ensures that USB device numbers are translated between the client device number and the server device number. Additionally, USB requests such as `SET_ADDRESS` and `CLEAR_HALT` are intercepted so that data maintained by the USB Core Driver can be updated correctly.

Transmission of all URBs over the IP network is via the TCP protocol. However, to avoid buffering delays and transmit the TCP/IP packets as soon as possible, the Nagle algorithm is disabled. The current USB/IP implementation does not use UDP for any communication. This is because the characteristics of transmission errors for USB and UDP/IP are quite different. Though the host controller does not resubmit failed Isochronous transactions, USB PDDs and devices expect that most transactions succeed. Also, transaction failures seldom occur in USB unless there is some physical problem with devices or cables. Therefore, in general, the transport layer for URBs must guarantee in order data arrival and retransmit lost packets.

The current implementation of USB/IP supports the Linux Kernel 2.6 series. The VHCI driver and the Stub driver are implemented as loadable kernel modules. Tools used to negotiate requested devices and set up TCP/IP connections are all in userland.

As a practical application of USB/IP, we have also developed a device sharing system for LAN environments (Figure 5). A key design of the device sharing system is the support of multiple virtual buses in the operating system. A virtual bus is implemented for each sophisticated peripheral bus. For service discovery, Multicast DNS [3] and DNS Service Discovery [2] are used to manage the dynamic device name space in a local area network. An example usage of this application is illustrated in Figure 6. This paper will focus primarily on the implementation of USB/IP. Details of the application of USB/IP for device sharing will be given in another paper.

4 Evaluation

In this section, we describe the characteristics of the USB/IP implementation. In particular, we show the results of several experiments that were carried out to measure the USB/IP performance. The computers used for the evaluation are listed in Table 1. To emulate various network conditions, we used the NIST Net [1] package on Linux Kernel 2.4.18. A client machine and a server machine were connected via a NIST Net machine, as shown in Figure 7. Both the client and server machines

Table 1: Machine Specifications for Experiments

CPU	Intel Pentium III 1GHz
Memory	SDRAM 512MB
NICs (client/server)	NetGear GA302T
NICs (NIST Net)	NetGear GA620
USB 2.0 Interface	NEC μPD720100

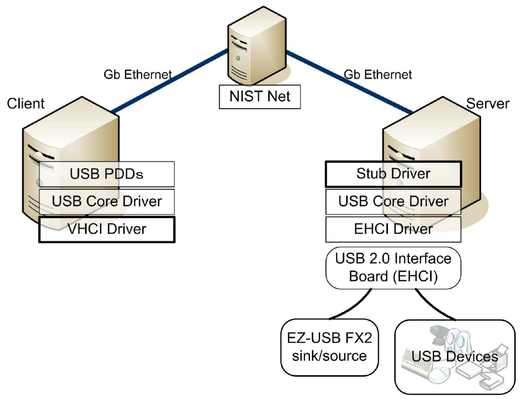


Figure 7: Experiment Environment

run Linux Kernel 2.6.8 with the USB/IP kernel modules installed.

4.1 Performance Evaluation of USB Pure Sink/Source

In the first evaluation, we used a pure sink/source USB device rather than a real USB device in order to identify the performance characteristics of USB/IP itself. A USB peripheral development board with a Cypress Semiconductor EZ-USB FX2 chip [5] was programmed to be a sink/source for the USB data. The firmware and test device driver were implemented as a USB PDD for the experiments.

4.1.1 Bulk Transfer

The I/O performance of USB/IP depends to a large degree on network delay and the data size of the operation being performed. In the second evaluation, we derived a throughput model for USB/IP from the results of the experiments, and then determined a set of criteria for optimizing USB/IP performance.

The first step of the evaluation was to measure the USB/IP overhead for USB requests of the Bulk transfer type. As Figure 8 shows, the test driver submits a Bulk URB to the remote source USB device, waits for the request to be completed, and then resubmits the request continuously. As shown in the figure, the enqueued URBs are transferred to the server's HCD between 1 and 2, and vice versa between 3 and 4. The execution time

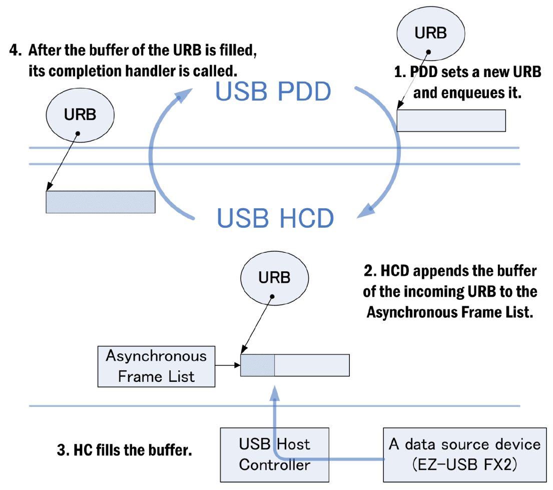

URB
4. After the buffer of the URB is filled, its completion handler is called.
1. PDD sets a new URB and enqueues it.
USB PDD
URB
USB HCD
URB
2. HCD appends the buffer of the incoming URB to the Asynchronous Frame List.
Asynchronous Frame List
3. HC fills the buffer.
USB Host Controller
A data source device (EZ-USB FX2)

Figure 8: Summarized Driver Behavior of the Experiments in Section 4.1.1

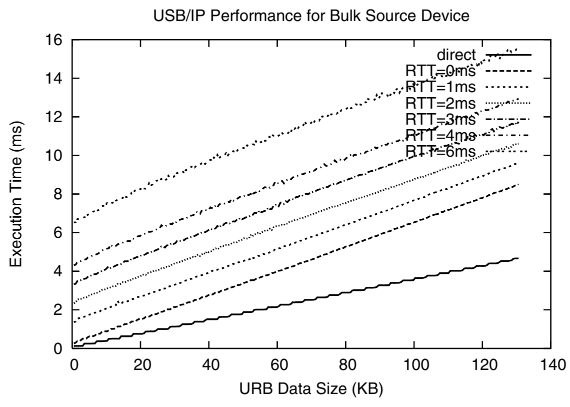


Figure 9: Execution Time of a URB

for processing a URB in the client for various values of URB data size and network round trip time (RTT) was measured using the Time Stamp Counter (TSC) register available on Intel Pentium processors. Note that when NIST Net sets the network RTT to 0ms, the actual RTT between the client and server machines is 0.12ms, as determined by `ping`.

The results are shown in Figure 9. From the graph, it can be seen that the relationship between the execution time of URBs and different data sizes is linear with constant gradient. The CPU cost for the USB/IP encapsulation is quite low at only a few percent. TCP/IP buffering does not influence the results because we use the `TCP_NODELAY` socket option. From the graph, the execution time t_{overIP} for data size s is given by

$$t_{overIP} = a_{overIP} \times s + RTT.$$

where a_{overIP} is the gradient value for the different

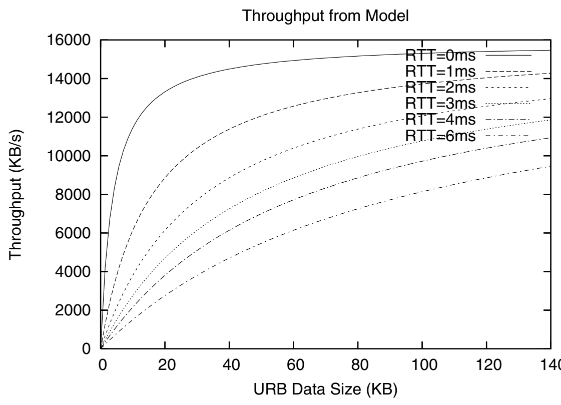


Figure 10: Throughput from Model

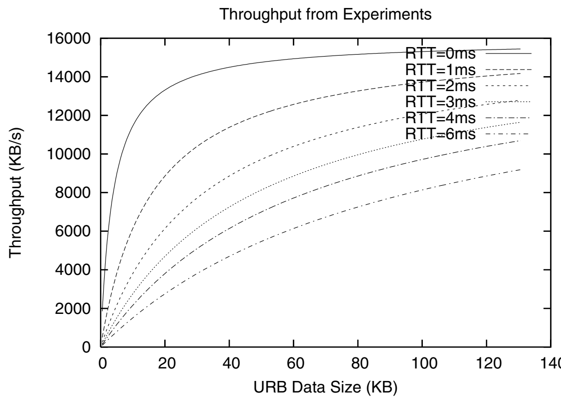


Figure 11: Throughput from Experiments

RTTs. The throughput *thpt* is then

$$thpt = \frac{s}{t_{overIP}} = \frac{s}{a_{overIP} \times s + RTT}. \quad (1)$$

A regression analysis shows a_{overIP} is 6.30e-2 ms/KB with a y-intercept for the 0ms case of 0.24 ms. The throughput modeled by Equation (1) is shown in Figure 10. The actual throughput from the experiments is illustrated in Figure 11, showing that the throughput of the model is fully substantiated by the experimental results. Therefore, within the parameter range of the experiments, this model is an accurate estimate of throughput for different URB data sizes and network RTTs.

In the directly-attached case, a_{direct} is 3.51e-2 ms/KB with a y-intercept of 0.07ms. This implies a relatively constant throughput of approximately 28.5MB/s except for quite small URB data sizes. This value is also determined largely by the performance of the host controller. The host controller in these experiments can process 6 or 7 Bulk I/O transactions per microframe (125us). In 480Mbps mode, one Bulk I/O transaction transfers 512B of data to a USB device. In this case, the throughput is equal to $(7 \times 512B)/125us = 28MB/s$.

To summarize these experiments, we have estimated

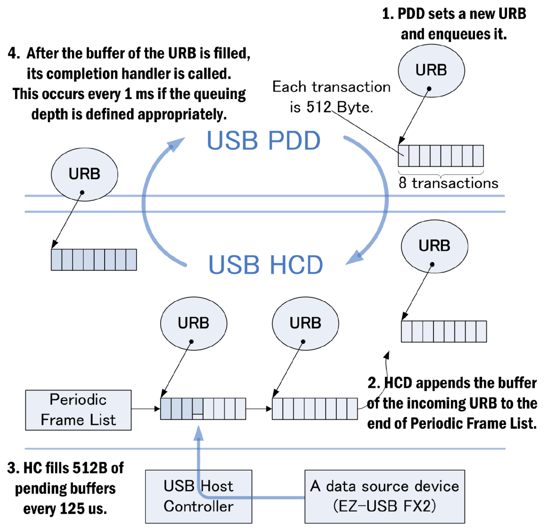

1. PDD sets a new URB and enqueues it.
4. After the buffer of the URB is filled, its completion handler is called. This occurs every 1 ms if the queuing depth is defined appropriately.
Each transaction is 512 Byte.
URB
USB PDD
8 transactions
USB HCD
URB
URB
URB
URB
Periodic Frame List
2. HCD appends the buffer of the incoming URB to the end of Periodic Frame List.
3. HC fills 512B of pending buffers every 125 us.
USB Host Controller
A data source device (EZ-USB FX2)

Figure 12: Summarized Driver Behavior of the Experiments in Section 4.1.2

the appropriate URB data size under various network delays. We have also confirmed that when multiple URBs were queued simultaneously, the throughput of Bulk transfer is dependent on the total data size of simultaneously-queued URBs. To maintain throughput when there is some network delay, USB PDDs should either enlarge each URB data size, or increase the queuing depth of URBs. Moreover, in situations when a large number of URBs are queued asynchronously, and there is substantial network delay, the TCP/IP window size must also be increased to ensure that the network pipe remains full.

4.1.2 Isochronous Transfer

In this section, we examine the performance of USB/IP when employing the isochronous transfer type. To ensure transfers for USB devices meet the Isochronous requirements, starvation of transaction requests must be avoided in the host controller. To achieve this, the USB driver model allows USB PDDs to queue multiple URBs simultaneously. In the case of USB/IP, it is also important to select a URB queuing depth that matches the likely delays introduced by the IP network.

For this test, we developed the firmware and a test device driver for an Isochronous source device. The device and drivers are configured as follows:

- A transaction moves 512B data in one microframe (125us).
- A URB represents 8 transactions.

In this case, the completion handler of the URB is called every 1ms ($125us \times 8$). This 1ms interval was chosen to be small enough so that it would be possible to examine the isochrony of USB/IP. In general, the interval of completion is set to approximately 10ms, which is an acceptable trade-off between smoothness of I/O operations and the processing cost. Figure 12 shows the detail of the driver behavior in the experiments. The USB PDD sets up each URB with the pointer to an I/O buffer for 8 transactions, and then queues the multiple URBs. The host controller then keeps pending I/O buffers on the Periodic Frame List, and the completion handler is called every 1ms. The HCD moves the input data to the USB PDD periodically. If there are no pending I/O buffers in the Periodic Frame List, the host controller does not copy data and isochronous data will be lost.

For a directly-attached source device, and the USB PDD submitting only one URB, the completion interval was 11.1ms because of request starvation. When the USB PDD submitted 2 or more URBs simultaneously, the completion intervals were 1ms, with a standard deviation of approximately 20ns for any queuing depth.

Figure 13 illustrates the mean completion intervals for various network RTTs and the queuing depths of submitted URBs for USB/IP. This shows that even under some network delays, the USB PDD is able to achieve 1ms completion intervals, provided that an adequate queuing depth is specified. For example, when the network delay is 8ms, the appropriate queuing depth is 10 or more. Figure 14 shows the time series of completion intervals in this case. Immediately after the start of the transfer, the completion intervals vary widely because the host controller does not have enough URBs to avoid starvation. Once enough URBs are available, the cycle becomes stable and the completion intervals remain at 1ms. Figure 15 shows the standard deviations of the completion intervals. With the sufficient URBs, the measured standard deviation is less than 10us, including the NIST Net's deviations [1]. These values are less than one microframe interval (125us) and adequate for most device drivers. This is the case for the Linux kernel 2.6 series, where process scheduling is driven by `jiffies`, which are incremented every 1ms. TCP/IP buffering has no impact on the timing, since the socket option `TCP_NODELAY` is set.

USB/IP periodic transfers are illustrated in Figure 16. In this case, the USB PDD in the client host queues 3 URBs which are completed every x ms. For the corresponding periodic completions in the server, the host controller must always keep multiple URBs queued. Therefore, with a queuing depth q, the time in which the next URB is pending is

$$t_{npending} = (q-1)x - RTT > 0.$$

The appropriate queuing depth q can then be calculated

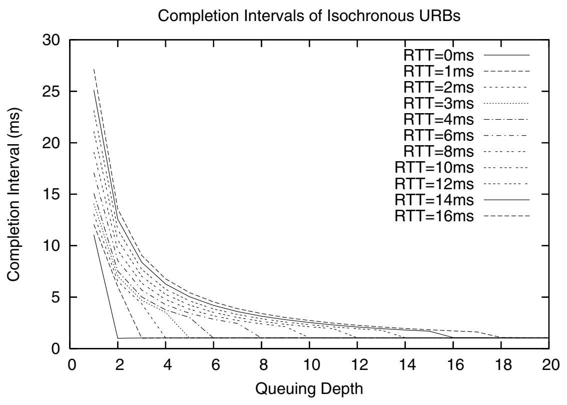


Figure 13: Mean Completion Intervals

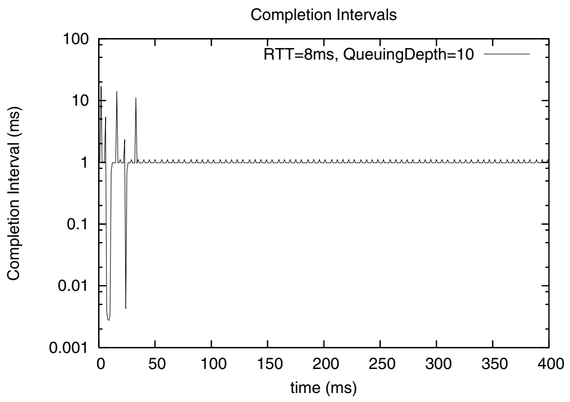


Figure 14: Time Series Data of Completion Intervals (RTT=8ms, Queuing Depth=10)

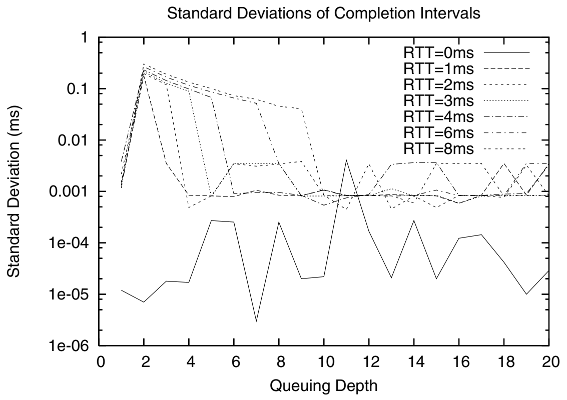


Figure 15: Standard Deviations of Completion Intervals

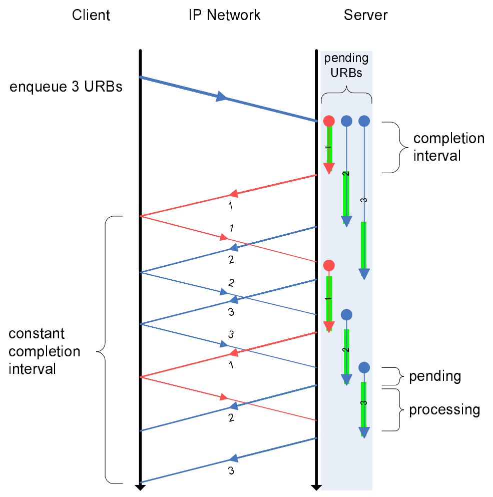


Figure 16: USB/IP Model for Periodical Transfers

by

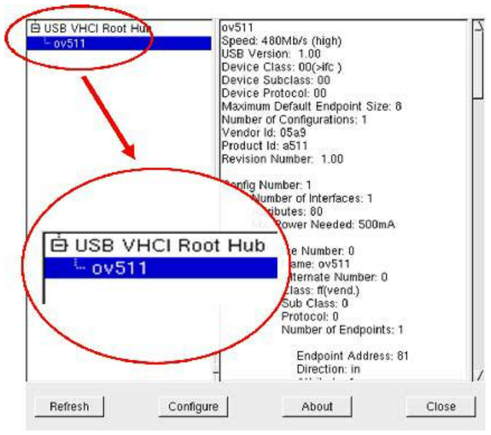
$$q > \frac{RTT}{x} + 1. \tag{2}$$

Comparing Figure 13 and Equation (2) shows the required queuing depth of URBs in the experiments.

To summarize these experiments, USB PDDs with periodic transfers must queue multiple URBs to at least the depth q of Equation (2) to ensure a continuous stream of I/O. In the wide area networks where there is significant jitter or packet loss, q should be increased to ensure a sufficient margin is available. The result can be also applied to Interrupt transfer type URBs, which specify the maximum delay of completion. This examination continues for common USB devices over an IP network in Section 4.2.2.

4.2 Performance Evaluation of USB Devices

In this section, we examine the USB/IP characteristics for common USB devices. All USB devices we have tested can be used as USB/IP devices. Figure 17 shows a client host attached to a remote USB camera through the VHCI driver. In this case, the USB device viewer `usbview` [10] sees the device descriptors as if the camera were locally attached. The only difference that is apparent between USB and USB/IP is that the host controller is VHCI. USB PDDs can also control their corresponding remote USB devices without any modification. In our LAN environment, the performance degradation of USB/IP is negligible. Specific details of the performance of each kind of USB/IP device are described in more detail below.



```
ov511
Speed: 480Mb/s (high)
USB Version:  1.00
Device Class: 00(>ifc )
Device Subclass: 00
Device Protocol: 00
Maximum Default Endpoint Size: 8
Number of Configurations: 1
Vendor Id: 05a9
Product Id: a511
Revision Number:  1.00

onfig Number: 1
   umber of Interfaces: 1
   ibutes: 80
   ower Needed: 500mA

    e Number: 0
    ame: ov511
    lternate Number: 0
    lass: ff(vend.)
    Sub Class: 0
    Protocol: 0
    Number of Endpoints: 1

       Endpoint Address: 81
       Direction: in
```

Figure 17: `usbview` Output for a USB/IP Device

Table 2: Specifications of the Tested USB Hard Disk

Product Name	IO-DATA HDA-iU120
Interface	USB 2.0/1.1, iConnect
Capacity	120GB
Rotation Speed	5400rpm
Cache Size	2MB

4.2.1 USB Bulk Device

USB storage devices (e.g., hard disks, DVD-ROM drives, and memory drives), USB printers, USB scanners and USB Ethernet devices all use the USB Bulk transfer type. All these devices are supported by USB/IP. For USB storage devices, it is possible to create partitions and file systems, perform mount/umount operations, and perform normal file operations. Moreover, we have shown that it is possible to play DVD videos and to write DVD-R media in a remote DVD drive, using existing, unmodified, applications. As described in Section 4.1.1, the USB/IP performance of USB Bulk devices depends on the queuing strategy of Bulk URBs. We have tested the original USB storage driver of Linux Kernel 2.6.8 to show its effectiveness for USB/IP.

The experimental setup for this test was the same as that described in Section 4.1.1. NIST Net was used to emulate various network delays. We used the Bonnie++ 1.03 [4] benchmarks for ext3 file system on a USB hard disk. The disk specifications are shown in Table 2. The Bonnie++ benchmarks measure the performance of hard drives and file systems using file I/O and creation/deletion tests. The file I/O tests measure sequential I/O per character and per block, and random seeks. The file creation/deletion tests execute `creat/stat/unlink` file system operations on a large number of small files.

Figure 18 and Figure 19 show the sequential I/O throughput and the corresponding CPU usage for USB and USB/IP respectively. Figure 20 shows sequential

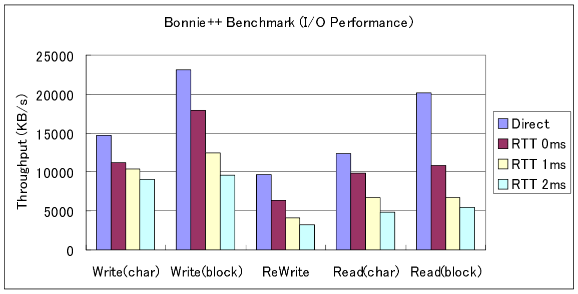


Figure 18: Bonnie++ Benchmark (Sequential Read/Write Throughput on USB and USB/IP)

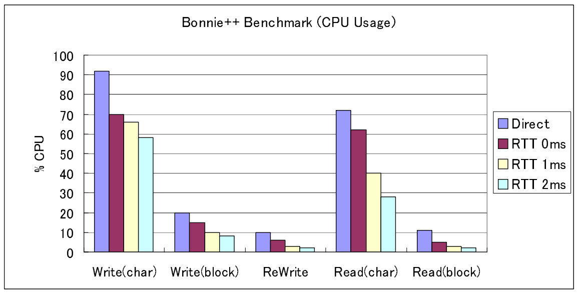


Figure 19: Bonnie++ Benchmark (Sequential Read/Write CPU Usage on USB and USB/IP)

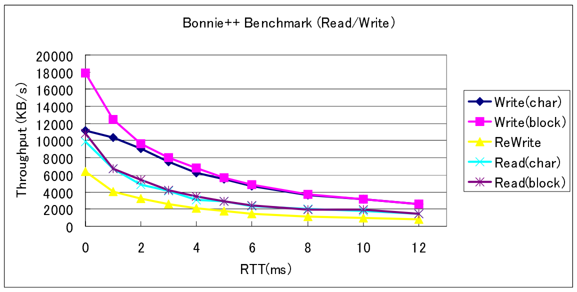


Figure 20: Bonnie++ Benchmark (Sequential Read/Write Throughput on USB/IP)

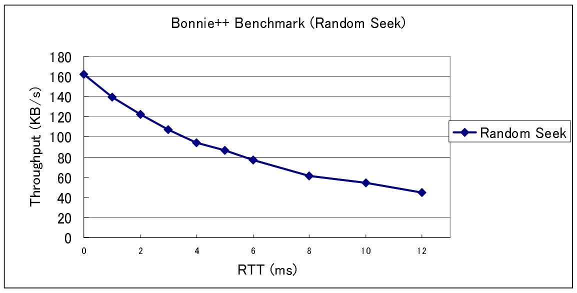


Figure 21: Bonnie++ Benchmark (Random Seek Speed on USB/IP)

I/O throughput by USB/IP under various network delays. For character write and block write, the throughput obtained by USB/IP when NIST Net's RTT is 0ms (0.12ms by `ping`) is approximately 77% of the throughput obtained by USB. Rewrite speeds are 66%, character read 79%, and block read 54% of that obtained by USB respectively. Since the CPU usage for USB/IP is less than those for USB, the bottleneck primarily results from insufficient queuing data size for the I/Os to the remote hard disk.

The Linux USB storage driver is implemented as a glue driver between the USB and SCSI driver stacks. For the SCSI stack, the USB storage driver is a SCSI host driver. A SCSI request with scatter-gather lists is repacked into several URBs, which are responsible for each scatter-gather buffer. The Linux USB storage driver does not support the queuing of multiple SCSI requests. Therefore, the total I/O data size of URBs submitted simultaneously is the same as each SCSI request size. In the case of block write, this is approximately 128KB. This queuing data size is small for USB/IP under some network delays, as we discussed in Section 4.1.1. To optimize the sequential I/O throughput for USB/IP, a reasonable solution is for the USB storage driver to provide SCSI request queuing.

The throughput of random seek I/O by USB/IP is illustrated in Figure 21. This test runs a total of 8000 random `lseek` operations on a file, using three separate processes. Each process repeatedly reads a block, and then writes the block back 10% of the time. The throughput obtained by USB for this test is 167KB/s. The throughput difference between USB and USB/IP is much smaller than that of sequential I/Os. The CPU usage in both the USB and USB/IP cases is 0%. This is because the bottleneck for random seek I/O is the seek speed of the USB hard disk itself, and is slower than that of read/write I/Os. The rational speed of the USB hard disk we tested is 5400rpm and its seek speed is approximately 10ms.

Figure 22 shows the speed of file creation and deletion operations by USB/IP under various network delays.

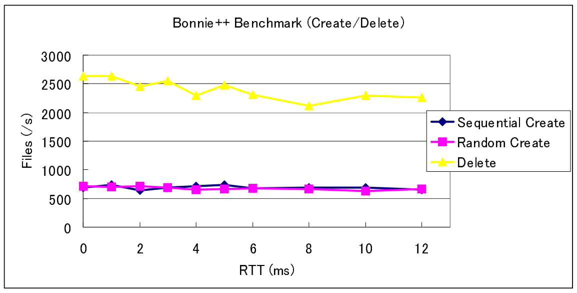


Figure 22: Bonnie++ Benchmark (Create/Delete Speed on USB/IP)

The speeds obtained by USB are 682 op/s for sequential creation, 719 op/s for random creation, and 2687 op/s for file deletion. In all these cases, the CPU usage was over 90%. The difference between USB and USB/IP is quite small, and the results of each test are almost constant under various network delays. The bottleneck for the file creation/deletion tests is predominantly the CPU resources.

4.2.2 USB Isochronous Device

USB multimedia devices, such as USB cameras and USB speakers, use the USB Isochronous transfer type to transmit data at periodic intervals. We have tested a USB camera (which uses the OmniVison OV511 chip), and a USB Audio Class speaker. These devices work completely transparently using USB/IP on a LAN. We were able to demonstrate video capture from the camera, and successfully played music using the speaker system.

The Linux USB Audio Class driver employs multi-buffering by submitting 2 URBs simultaneously, where each URB is responsible for 5ms of transactions. Equation (2) shows this driver will work with a remote USB audio device provided that the network delay is 5ms or less. For larger network delays, it is still possible to use a remote audio device by increasing the completion interval for each URB. The main drawback with this is that it can result in degraded I/O response.

4.2.3 USB Interrupt Device

USB Human Input Devices, such as USB keyboards and USB mice, use the USB Interrupt transfer type to transmit data at periodic intervals, in a similar manner to interrupt requests (IRQs). Other devices also use the Interrupt transfer type to notify hosts of status changes. On our test LAN, we were able to demonstrate the correct operation of such devices using USB/IP for both consoles and the X Window System.

Most USB HID drivers submit only one URB with a completion delay of 10ms. After the URB processing is completed, the driver resubmits the URB. The drivers read the interrupt data, which is accumulated in the device endpoint buffer, every 10ms. Under large network delays, it is possible that the device endpoint buffer may overflow. When network delays approach 100ms, there is likely to be significant degradation in the performance of these devices. The former problem can be resolved by queuing more URBs so that endpoint buffer overflow is prevented. The latter problem is an underlying issue that results from attempting human interaction over a network.

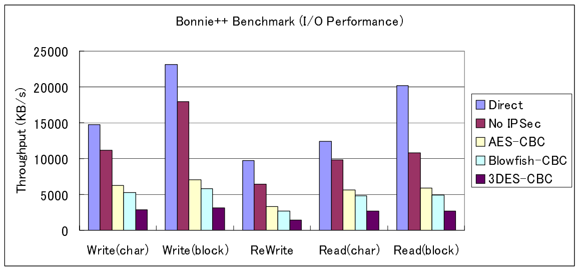


Figure 23: Bonnie++ Benchmark (Sequential Read/Write Throughput on USB and USB/IP with IPSec)

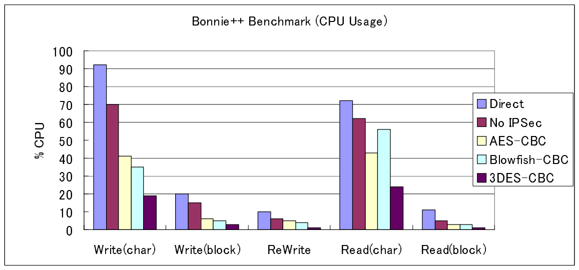


Figure 24: Bonnie++ Benchmark (Sequential Read/Write CPU Usage on USB and USB/IP with IPSec)

5 Discussion

Security. Support for authentication and security schemes is an important part of the USB/IP architecture. As most of USB/IP is currently implemented in the kernel (to avoid memory copy overhead), it is logical that an existing kernel-based security mechanism be employed. IPSec [9], which provides a range of security services at the IP layer, is one of the most suitable technologies for this purpose. IPSec provides the following functionality: access control, connectionless integrity, data origin authentication, protection against replay attacks (a form of partial sequence integrity), confidentiality (encryption), and limited traffic flow confidentiality. Figure 23 and Figure 24 show the results of the I/O benchmark described in Section 4.2.1, but with IPSec ESP (Encapsulating Security Payload) employed for all traffic between the client and the server. The hash algorithm used is HMAC-SHA1 and the encryption algorithms are AES-CBC, Blowfish-CBC, and 3DES-CBC, respectively. Since encryption entails a significant CPU overhead, the performance degradation compared to non-encrypted transfer is quite high. In the case of AES-CBC encryption, the performance is less than 50% of that obtained without IPSec. To achieve significantly higher throughput, such as would be necessary for embedded computers, some form of hardware acceleration is required. The optimization criteria, described in Section 4.1.1, may also be applied to determine the likely impact of IPSec overhead on the operation of USB/IP. To do this, IPSec overhead is considered as an estimate of pseudo network delay. The issue of authentication and security for USB/IP is an ongoing area of research, and will be dealt with more fully in a future paper.

Error Recovery. The issue of error recovery is an area that also needs to be considered for USB/IP. The error recovery methodology employed by USB/IP exploits the semantics of error recovery in the USB protocol. A dropped TCP/IP connection to a remote USB device is detected by both the VHCI driver and the Stub driver. The VHCI driver detaches the device, so it appears that the device has been disconnected, and the Stub driver resets the device. As with directly-attached USB devices that are disconnected, some applications and drivers may lose data. This recovery policy is appropriate in LAN environments, because sudden disconnection of TCP/IP sessions seldom occur. In addition, this error recovery policy greatly simplifies the USB/IP implementation. To use USB/IP with more unstable network environments, such as mobile networks, a more dependable recovery scheme is required. This is also an area of future research.

Interoperability. The USB/IP architecture has been designed so that it is interoperable between different operating systems. USB driver stacks for many operating systems are very similar. In most cases the USB driver stacks are designed with three layers: the USB PDDs, the USB HCDs, and a glue layer. In addition, the Linux, Microsoft Windows, and FreeBSD operating systems also have similar USB request structures, (`urb`, `URB` and `usbd_xfer` respectively), which means that USB/IP will be relatively easy to port. One area that this paper does not focus on is details of the interoperability features of the USB/IP protocol itself. Future work will be required to develop the interoperable implementations for these other operating systems.

6 Related Work

iSCSI [14] is designed to transport SCSI packets over a TCP/IP network, and provide access to remote storage devices. This protocol is commonly regarded as a fundamental technology for the support of SANs (Storage Area Networks). However, because iSCSI is the extension of a SCSI bus over an IP network, the protocol has been designed to ensure network transparency and interoperability. This means that the protocol could be applied to storage device sharing between computers using our peripheral bus extension technique. Our device sharing methodology is designed to provide virtual connections that are independent of device control details, so it would be possible to utilize the iSCSI protocol, as one of the control modules for the remote device. The main limitation of iSCSI is that it supports only storage devices. USB/IP has the advantage that all types of devices, including isochronous devices, can be controlled over IP networks.

University of Southern California's Netstation [6] is a heterogeneous distributed system composed of processor nodes and network-attached peripherals. The peripherals (e.g., camera, display, emulated disk, etc.) are directly attached to a shared 640Mbps Myrinet or to a 100Mbps Ethernet. The goal of Netstation is to share resources and improve system configuration flexibility. In this system, VISA (Virtual Internet SCSI Adapter) [11] emulates disk drives using UDP/IP. This project is similar to our peripheral bus extension, as both systems use IP network to transfer data [7]. However, while Netstation is a network-based computer architecture that allows easy substitution of systems, our architecture aims to share already-attached devices between heterogeneous computers. We believe it offers the most practical approach to exploiting today's sophisticated peripheral buses and their device drivers.

The Inside Out Network's AnywhereUSB [8] is a network-enabled USB hub. This hub employs proprietary USB over IP technology, and provides remote access to USB devices attached to ports on the hub, though in a somewhat limited manner. The hub supports only USB Bulk and Interrupt devices operating at 12Mbps in a LAN environment. Most USB storage devices, which operate at 480Mbps, and USB isochronous devices are not supported. In contrast, USB/IP supports all types of USB devices operating at up to 480Mbps. Our evaluation has shown that, in LAN environments, all the tested devices work perfectly with the original PDD. Moreover, we have shown that there is an optimization strategy that will enable USB/IP to operate effectively even under larger network delays.

There are a number of network appliances which export the resources of specific USB devices to an IP network. Some NAS appliances share their attached USB storages via NFS or CIFS. In addition, some home routers have USB ports that can be used to share connected USB printers or USB webcams. As we have described in Section 2.1, these appliances, which employ coarse-grained protocols, do not support the low-level operations which a required to allow the remote devices to operate in a transparent manner.

A new technology which is under development is Wireless USB [16], which employs UWB (Ultra Wide Band) to provide expanded USB connectivity. This technology aims to eliminate USB cables altogether, however the effective communication range is limited to 10 meters. The implementation of Wireless USB is very similar to the physical layer of USB, so this technology will complement USB/IP. This will allow USB/IP to be used for Wireless USB devices, and enabling USB/IP to provide remote device access with virtually any IP network infrastructure.

7 Conclusion

We have developed a peripheral bus extension over IP networks, that provides an advanced device sharing architecture for the support of recent sophisticated peripheral bus interfaces. The device sharing architecture meets a range of functionality requirements, including network transparency, interoperability, and generality, by utilizing the low-level device control protocols of the peripheral interfaces.

As a practical example of the peripheral bus extension, we designed and implemented USB/IP, which allows a range of remote USB devices to be used from existing applications without any modification of the application or device drivers. We also undertook a range of experiments to establish that the I/O performance of remote USB devices connected using USB/IP is sufficient for actual usage. We also determined the performance characteristics of USB/IP, and developed optimization criteria for IP networks.

There are three primary design criteria that need to be considered in order to effectively support the transfer of fine-grained device control operations over IP networks. First, for asynchronous devices (known as bulk devices), the device driver must queue enough request data to ensure maximum throughput for remote devices. Second, synchronous devices (known as isochronous devices), require a smooth of I/O stream. To achieve this, the appropriate number of requests must be queued to avoid starvation of requests at the physical device. However, this is a trade-off, because a large queue size reduces the response of each device. Finally, most situations require the response to a request to arrive within a certain time. In some situations, it is possible to relax the restriction by modifying a device driver or an application.

8 Availability

The USB/IP implementation is available under the open source license GPL. The information is at `http://usbip.naist.jp/`.

References

[1] Mark Carson and Darrin Santay. NIST Net: A Linux-Based Network Emulation Tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[2] Stuart Cheshire and Marc Krochmal. DNS-Based Service Discovery. Internet Draft, draft-cheshire-dnsext-dns-sd.txt, Feb 2004.

[3] Stuart Cheshire and Marc Krochmal. Performing DNS Queries via IP Multicast. Internet Draft, draft-cheshire-dnsext-multicastdns.txt, Feb 2004.

[4] Russell Coker. Bonnie++. `http://www.coker.com.au/bonnie++/`.

[5] Cypress Semiconductor Corporation. EZ-USB FX2. `http://www.cypress.com/`.

[6] Gregory G. Finn and Paul Mockapetris. Netstation Architecture: Multi-Gigabit Workstation Network Fabric. In *Proceedings of Interop Engineers' Conference*, 1994.

[7] Steve Hotz, Rodney Van Meter, and Gregory G. Finn. Internet Protocols for Network-Attached Peripherals. In *Proceedings of the Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in Cooperation with Fifteenth IEEE Symposium on Mass Storage Systems*, Mar 1998.

[8] Inside Out Networks. AnywhereUSB. `http://www.ionetworks.com/`.

[9] Stephen Kent and Randall Atkinson. Security Architecture for the Internet Protocol. RFC2406, Nov 1998.

[10] Greg Kroah-Hartman. usbview. `http://sf.net/projects/usbview/`.

[11] Rodney Van Meter, Gregory G. Finn, and Steve Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. *ACM SIGOPS Operating System Review*, 32(5):71–80, Dec 1998.

[12] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[13] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, and Kang Yueh. RFS Architectural Overview. In *USENIX Conference Proceedings*, pages 248–259, Jun 1989.

[14] Julian Satran, Kalman Meth, Costa Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC3720, Apr 2004.

[15] Sun Microsystems, Inc. NFS: Network File System Protocol Specification. RFC1094, Mar 1989.

[16] Wireless USB Promoter Group. `http://www.usb.org/wusb/home/`.

Trickle: A Userland Bandwidth Shaper for Unix-like Systems

Marius A. Eriksen[*]
Google, Inc.
`mae@google.com`

Abstract

As with any finite resource, it is often necessary to apply policies to the shared usage of network resources. Existing solutions typically implement this by employing traffic management in edge routers. However, users of smaller networks regularly find themselves in need of nothing more than ad-hoc rate limiting. Such networks are typically unmanaged, with no network administrator(s) to manage complicated traffic management schemes. Trickle bridges this gap by providing a simple and portable solution to rate limit the TCP connections of a given process or group of processes.

Trickle takes advantage of the Unix dynamic loader's preloading functionality to interposition itself in front of the BSD socket API provided by the system's `libc`*. Running entirely in user space, shapes network traffic by delaying and truncating socket I/Os without requiring administrator privileges. Instances of Trickle can cooperate, even across networks allowing for the specification of global rate limiting policies. Due to the prevalence of BSD sockets and dynamic loaders, Trickle enjoys the benefit of portability accross a multitude of Unix-like platforms.*

1 Introduction

Bandwidth shaping is traditionally employed monolithically as part of network infrastructure or in the local operating system kernel which works well for providing traffic management to large networks. Such solutions typically require dedicated administration and privileged access levels to network routers or the local operating system.

Unmanaged network environments without any set bandwidth usage policies (for example home and small office networks) typically do not necessitate mandatory traffic management. More likely, the need for bandwidth shaping is largely ad-hoc, to be employed when and where it is needed. For example,

- bulk transfers may adversely impact an interactive session and the two should receive differentiated services, or

- bulk transfers may need to be prioritized.

Furthermore, such users may not have administrative access to their operating system(s) or network infrastructure in order to apply traditional bandwidth shaping techniques.

Some operating systems provide the ability to shape traffic of local origin (these are usually extensions to the router functionality provided by the OS). This functionality is usually embedded directly in the network stack and resides in the operating system kernel. Network traffic is not associated with the local processes responsible for generating the traffic. Rather, other criteria such as IP, TCP or UDP protocols and destination IP addresses are used in classifying network traffic for shaping. These policies are typically global to the host (thus applying to all users on it). Since these policies are mandatory and global, it is the task of the system administrator to manage the traffic policies.

These are the many burdens that become evident if one would like to employ bandwidth shaping in an ad-hoc manner. While there have been a few attempts to add voluntary bandwidth shaping capabilities to the aforementioned in-kernel shapers[25], there is still a lack of a viable implementation and there is no use of collaboration between multiple hosts. These solutions are also non-portable and there is a lack of any standard application or user interfaces.

We would like to be able to empower any unprivileged user to employ rate limiting on a case-by-case basis, without the need for special kernel support. Trickle addresses precisely this scenario: Voluntary ad-hoc rate

[*]Work done by the author while at the University of Michigan.

limiting without the use of a network wide policy. Trickle is a portable solution to rate limiting and it runs entirely in user space. Instances of Trickle may collaborate with each other to enforce a network wide rate limiting policy, or they may run independently. Trickle only works properly with applications utilizing the BSD socket layer with TCP connections. We do not feel this is a serious restriction: Recent measurements attribute TCP to be responsible for over 90% of the volume of traffic in one major provider's backbone[16]. The majority of non-TCP traffic is DNS (UDP) – which is rarely desirable to shape anyway.

We strive to maintain a few sensible design criteria for Trickle:

- *Semantic transparency*: Trickle should never change the behavior or correctness of the process it is shaping (Other than the data transfer rates).
- *Portability*: Trickle should be extraordinarily portable, working with any Unix-like operating system that has shared library and preloading support.
- *Simplicity*: No need for excessively expressive policies that confuse users. Don't add features that will be used by only 1 in 20 users. No setup cost, a user should be able to immediately make use of Trickle after examining just the command line options (and there should be very few command line options).

The remainder of this paper is organized as follows: Section 2 describes the linking and preloading features of modern Unix-like systems. Section 3 provides a high-level overview of Trickle. Section 4 discusses the details of Trickle's scheduler. In section 5 we discuss related work. Finally, section 9 concludes.

2 Linking and (Pre)Loading

Dynamic linking and loading have been widely used in Unix-like environments for more than a decade. Dynamic linking and loading allow an application to *refer* to an external symbol which does not need to be resolved to an address in memory until the runtime of the particular binary. The canonical use of this capability has been to implement *shared libraries*. Shared libraries allow an operating system to share one copy of commonly used code among any number of processes. We refer to resolving these references to external objects as *link editing*, and to unresolved external symbols simply as *external symbols*.

After compilation, at link time, the linker specifies a list of libraries that are needed to resolve all external symbols. This list is then embedded in the final executable file. At load time (before program execution), the link editor maps the specified libraries into memory and resolves all external symbols. The existence of any unresolved symbols at this stage results in a run time error.

To load load its middleware into memory, Trickle uses a feature of link editors in Unix-like systems called preloading. Preloading allows the user to specify a list of shared objects that are to be loaded together the shared libraries. The link editor will first try to resolve symbols to the preload objects (in order), thus selectively bypassing symbols provided by the shared libraries specified by the program. Trickle uses preloading to provide an alternative version of the BSD socket API, and thus socket calls are now handled by Trickle. This feature has been used chiefly for program and systems diagnostics; for example, to match `malloc` to `free` calls, one would provide an alternative of these functions via a preload library that has the additional matching functionality.

In practice, this feature is used by listing the libraries to preload in an environment variable. Preloading does not work for set-UID or set-GID binaries for security reasons: A user could perform privilege elevation or arbitrary code execution by specifying a preload object that defines some functionality that is known to be used by the target application.

We are interested in interpositioning Trickle in between the shaped process and the socket implementation provided by the system. Another way to look at it, is that Trickle acts as a proxy between the two. We need some way to call the procedures Trickle is proxying. The link editor provides this functionality through an API that allows a program to load an arbitrary shared object to resolve any symbol contained therein. The API is very simple: Given a string representation of the symbol to resolve, a pointer to the location of that symbol is returned. A common use of this feature is to provide plug-in functionality wherein plugins are shared objects and may be loaded and unloaded dynamically.

Figure 1 illustrates Trickle's interpositioning.

3 How Trickle Works

We describe a generic rate limiting scheme defining a black-box scheduler. We then look at the practical aspects of how Trickle interpositions its middleware in order to intercept socket calls. Finally we discuss how multiple instances of Trickle collaborate to limit their aggregate bandwidth usage.

3.1 A Simple Rate Limiting Scheme

A process utilizing BSD sockets may perform its own rate limiting. For upstream limiting, the application can do this by simply limiting the rate of data that is written to a socket. Similarly, for downstream limiting, an application may limit the rate of data it *reads* from a socket. However, the reason why this works is not immediately obvious. When the application neglects to read some data from a socket, its socket receive buffers fill up. This in turn will cause the receiving TCP to advertise a smaller receiver window (`rwnd`), creating back pressure on the underlying TCP connection thus limiting its data flow. Eventually this "trickle-down" effect achieves end-to-end rate limiting. Depending on buffering in all layers of the network stack, this effect may take some time to propagate. More detail regarding the interaction between this scheme and TCP is provided in section 4.

While this scheme is practical, two issues would hinder widespread employment. Firstly, the scheme outlined is deceptively simple. As we will see in section 4, there are many details which make shaping at this level of abstraction complicated. The second issue is that there are no standard protocols or APIs for multiple processes to collaborate.

We also argue that employing bandwidth shaping inside of an application breaks abstraction layers. It is really the task of the operating system to apply policies to bandwidth usage, and it should not need to be a feature of the application. Even if libraries were developed to assist application developers, employing rate limiting in this manner would still put considerable burden on the developers and it should not be expected that every developer would even support it. The socket API provided by the OS provides certain functionality, and it should be the freedom of the application to use it unchanged, and not have to rely on semantics at the lower levels of abstraction in order to limit bandwidth usage.

There are also exceptions to these arguments. For example, certain protocols may benefit from application level semantics to perform shaping. Another example is that some applications may be able to instruct the sending party to limit its rate of outbound traffic[9] which is clearly preferable over relying on TCP semantics to perform traffic shaping.

Trickle provides a bandwidth shaping service without the need to modify applications. Trickle augments the operating system by *interpositioning* its middleware in front of the `libc` socket interface. From there, Trickle applies rate limiting to any dynamically linked binary that uses the BSD socket layer. By providing a standard command line utility, Trickle provides a simple and consistent user interface to specify rate limiting parameters. Communicating with the *trickle daemon*, allows all instances of Trickle to participate in collaborative rate limiting, even across hosts.

In addition to allowing portability, this approach offers several advantages. There is no need for extending

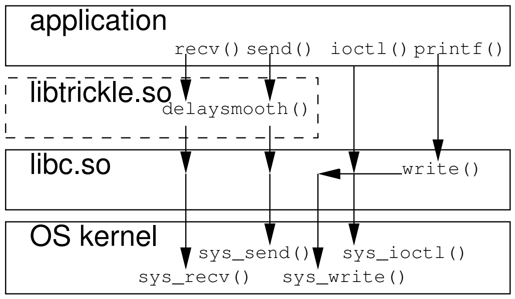


Figure 1: `libtrickle.so` is preloaded in the application's address space, calls to `recv()` and `send()` are handled by `libtrickle.so` and passed down to `libc`.

the kernel nor configuring such extensions; any user may use and configure Trickle any way she wants, making it ideal for ad-hoc rate limiting. There are also a number of advantages to this approach from the developer's point of view. Furthermore, being entirely contained in userland has made Trickle inherently easier to develop. It is easier to perform experiments and the software is easier to maintain and will be understood by a wider audience.

The primary disadvantage to using this approach is that all usage of Trickle is voluntary – that is, one cannot enforce rate limiting by policy (though some operating systems provide a mechanism for administrators to enforce preload libraries, there are still ways to get around its interpositioning). For its intended usage, this is not a big drawback as ad-hoc bandwidth shaping implies users do so voluntarily. Secondly and with smaller impact, Trickle cannot work with statically linked binaries.

3.2 The Mechanics of Library Interpositioning

With very rare exceptions, network software for Unix-like systems uses the socket abstraction provided by the operating system. In reality, the socket abstraction is entirely contained in the system call layer with corresponding `libc` shims[1]. Thus, with the use of the link editor's preload functionality, we interposition the Trickle middleware at a convenient level of abstraction and we do so entirely in user space.

Using preload objects, we replace the BSD socket abstraction layer provided by `libc`. However, to successfully interposition the Trickle middleware, we must be able to call the original version of the very interface we have replaced. To resolve this issue, we need to take advantage of the second feature of the link editor we discussed: We simply explicitly resolve the `libc` shims and call them as needed. This is done by opening the actual object file that contains `libc`, and using the link-editor API to resolve the symbols needed. The location of the `libc` shared object is discovered in the configuration/compilation cycle, but could just as easily be discovered dynamically at run time. Figure 1 attempts to illustrate the mechanics of the interpositioning of the Trickle middleware.

In practice, preload objects are specified by the environment variable `LD_PRELOAD`. Trickle's command line utility, `trickle`, sets this environment variable to the object that contains Trickle's middleware. Additionally, it passes any parameters specified by the user in other environment variables in a well defined namespace. These parameters may include upstream or downstream rates to apply, as well as whether or not this instance of Trickle should collaborate with other instances of Trickle.

3.3 The Life of a Socket

New sockets are created with either the `socket()` or `accept()` interfaces. An old socket is aliased with calls to `dup()` or `dup2()`. Any new or duplicated socket is marked by Trickle by keeping an internal table indexing every such socket. File descriptors that are not marked are ignored by Trickle, and relevant calls specifying these as the file descriptor argument are simply passed through to the `libc` shims without any processing. Note that it is also possible for an application to perform file descriptor passing: An application may send an arbitrary file descriptor to another over local inter process communication (IPC), and the receiving application may use that file descriptor as any other. File descriptor passing is currently not detected by Trickle. When a socket is closed, it is unmarked by Trickle. We say that any marked socket is *tracked* by Trickle.

Two categories of socket operations are most pertinent to Trickle: Socket I/O and socket I/O multiplexing. In the following discussion we assume that we possess a black box. This black box has as its input a unique socket identifier (e.g. file descriptor number) and the direction and length of the I/O operation to be performed on the socket. A priority for every socket may also be specified as a means to indicate the wish for differentiated service levels between them. The black box outputs a recommendation to either *delay* the I/O, to *truncate* the length of the I/O, or a combination of the two. We refer to this black box as the Trickle *scheduler* and it discussed in detail in a later section.

The operation of Trickle, then, is quite simple: Given a socket I/O operation, Trickle simply consults the scheduler and delays the operation by the time specified, and when that delay has elapsed, it reads or writes at most the number of bytes specified by the scheduler. If the socket is marked non-blocking, the scheduler will specify the length of I/O that is immediately allowable. Trickle will perform this (possibly truncated) I/O and return immediately, as to not block and violate the semantics of non-blocking sockets. Note that BSD socket semantics allow socket I/O operations to return short counts – that is, an operation is not required to complete in its entirety and it is up to the caller to ensure all data is sent (for example by looping or multiplexing over a calls to `send()` and `recv()`). In practice, this means that the Trickle middleware is also allowed to return short I/O counts for socket I/O operations without affecting the semantics of the socket abstraction. This is an essential property of the BSD socket abstraction that we use in Trickle.

Multiplexing I/O operations, namely calls to `select()` and `poll()`[2] are more complex. The purpose of the I/O multiplexing interface is to, given a set of file descriptors and conditions to watch for each, notify the caller when any condition is satisfied (e.g. file descriptor x is ready for reading). One or more of these file descriptors may be tracked by Trickle, so it is pertinent for Trickle to wrap these interfaces as well. Specifically, `select()` and `poll()` are wrapped, these may additionally wait for a *timeout* event (which is satisfied as soon as the specified timeout value has elapsed).

To simplify the discussion around how Trickle handles multiplexing I/O, we abstract away the particular interface used and assume that we deal only with a set of file descriptors, one or more of which may be tracked by trickle. Also specified is a global timeout. For every file descriptor that is in the set and tracked by Trickle, the scheduler is invoked to see if the file descriptor would be capable of I/O immediately. If it is not, it is removed from the set and added to a holding set. The scheduler also returns the amount of time needed for the file descriptor to become capable of I/O, the holding time. The scheduler calculates this on the basis of previously observed I/O rates on that socket. Trickle now recalculates the timeout to use for the multiplexing call: This is the minimum of the set of holding times and the global timeout.

Trickle then proceeds to invoke the multiplexing call with the new set of file descriptors (that is, the original set minus the holding set) and the new timeout. If the call returns because a given condition has been satisfied, Trickle returns control to the caller. If it returns due to a timeout imposed by Trickle, the process is repeated, with the global timeout reduced by the time elapsed since the original invocation of the multiplexing call (wrapper). In practice, a shortcut is taken here, where only file descriptors from the holding set are examined, and rolled in if ready. The process is repeated until any user specified condition is satisfied by the underlying multiplexing call.

3.4 Collaboration

We have detailed how Trickle works with a set of sockets in a single process, though more often than not it is highly practical to apply *global rate limits* to a set of processes that perform network I/O. These processes do not necessarily reside a single host; it is often useful to apply them on every process that contributes to the network traffic passing through a particular gateway router, or to use a global limit to control the utilization of the local area network. It may also be desirable to apply individual rate limiting policies for processes or classes of processes.

Trickle solves this by running a daemon, `trickled` which coordinates among multiple instances of Trickle. The user specifies to `trickled` the *global* rate limitations which apply across all instances of Trickle. That is, the aggregate I/O rates over all processes shaped by Trickle may not exceed the global rates. Furthermore, the user can specify a priority per instance or type of instance (e.g. interactive applications), allowing her to provide differentiated network services to the various client applications.

By default, `trickled` listens on a BSD domain socket and accepts connections from instances of Trickle running on the local host. These instances then request a bandwidth allocation from `trickled`. The bandwidth allocation is computed using the same black box scheduler described previously. It is used in a slightly different mode where the scheduler simply outputs the current rate the entity is assigned. These rate allocations may change frequently, and so the instances of Trickle get updated allocations with some preset regularity.

It is worth noting that there is a simple denial of service attack should a rogue user exist on the system. This user could simply create as many fake Trickle instances as necessary and, without actually doing any socket I/O, report data transfers to `trickled`. Of course, such a user could, though using more resources to do so, also consume as much network resources as possible, in effect achieving the same result by exploiting TCP fairness.

The same model of collaboration is applied across several hosts. Instead of listening on a Unix domain socket, `trickled` listens on a TCP socket, and can thus schedule network resource usage across any number of hosts. In this scenario, rate allocation updates may start to consume a lot of local network resources, so care must be taken when setting the frequency at which updates are sent.

4 I/O Scheduling With Rate Restrictions

The problem of rate limiting in Trickle can be generalized to the following abstraction: Given a number of entities capable of transmitting or receiving data, a global rate limit must be enforced. Furthermore, entities may have different *priorities* relative to each other as to differentiate their relative service levels. In Trickle, we use this abstraction twice: In a shaped process, a socket is represented as an entity with priority `1`. In `trickled` every collaborating process is represented by an entity (the collaborating processes may even reside on different hosts) and every entity is assigned a priority according to a user specified policy.

When an entity is ready to perform some I/O, it must consult the scheduler. The scheduler may then advise the entity to *delay* its request, to *partially complete* the request (i.e. truncate the I/O operation), or a combination of the two. In this capacity, the scheduler is *global* and coordinates the I/O allocation over all entities. In another mode, the scheduler simply outputs the current global rate allocation for the requesting entity.

After an entity has performed an I/O operation, it notifies the Trickle scheduler with the direction (sent or received) and length of the I/O. Trickle then updates a bandwidth statistics structure associated with that entity and direction of data. This structure stores the average data throughput rate for the entire lifetime of that entity as well as a windowed average over a fixed number of bytes. Also, an aggregate statistic covering all entities is updated.

Before an entity performs I/O, it consults the Trickle scheduler to see how much delay it must apply and how much data it is allowed to send or receive after the delay has elapsed. Let us assume for the moment that the scheduler need only decide for how long to delay the requested I/O operation.

4.1 Distribution and allocation

Every entity has an assigned number of *points* inversely proportional to that entity's priority. The global rate *limit* is divided by the total number of points over all entities, and this is the rate allotment per point. If every entity performed I/O with a rate equal to its number of points multiplied by the per point allotment, the total rate over all entities would be at the rate limit and every entity would perform I/O at a rate proportional to their assigned priority. Since Trickle is performing bandwidth shaping, most often the entities has the ability to exceed the transfer rates that they are assigned by the scheduler. The entities only very seldomly behave in any predictable manner: Their data transfer rates may be bursty, they may have consistent transfer rates lower than their alloted rates, or they might be idle. At the same time, the scheduler needs to make sure that the entities in aggregate may transfer data at a rate capped only by the total rate limit: Trickle should never hinder its client applications from fully utilizing the bandwidth allocated to them.

Statistics structures as well as limits are kept independently *per-direction* and thus Trickle is fully asymmetric. Furthermore, it is worthy to note that Trickle makes scheduling decisions based on a *windowed average*. Dealing with instantaneous bursts is discussed later.

4.2 Scheduling

Trickle uses a simple but effective algorithm to schedule I/O requests. It is a global scheduler that preserves every requirement outlines above. We maintain a total T, which is initialized with the total number of points alloted over all entities. We also maintain a per-point allotment P which is initially calculated as outlined above. An entity consumes bandwidth less than its allotment if its measured (windowed average) consumption is less than its number of points E_p multiplied by the per-point allotment P.

For every entity that consumes bandwidth less than its allotment, we subtract E_p from T, and then add the difference between E's actual consumption and it's aloted consumption to a free pool, F. After iterating over the entities, the value of the free pool is redistributed amongst the remaining entities. In practice, this is done by inflating the per-point allotment: $P = P + F/T$.

This process is then repeated until there are either *no remaining entities* that have more allotment than their consumption or *all remaining entities* have more allotment than their consumption. This is the stable state. We call the final allotment of each entity after this process the *adjusted allotment*.

After the adjustment process, if the entity being scheduled is currently consuming bandwidth at a rate less than its adjusted allotment, Trickle allows the operation to proceed immediately. If not, it requests the entity to delay the operation by the time it would take to send the requested number of bytes at the adjusted rate.

Figure 2 shows bandwidth consumption at the receiving end of two bulk transfers shaped collaboratively by Trickle, one having a lower priority. The aggregate bandwidth consumption (i.e. the sum of the two) is also shown. Trickle was configured with a global receive limit of 10 kB/s. The lower priority transfer has an average transfer rate of 3,984 bytes/second with a standard deviation of 180 bytes/second. The higher priority transfer averages at 5,952 bytes/sec with a standard deviation of 455 bytes/second.

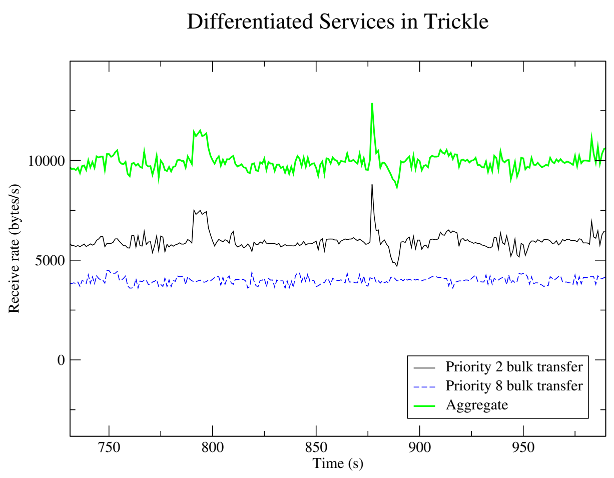


Figure 2: Measuring a windowed-average bandwidth consumption of two bulk transfers with differentiated service. Trickle was configured with a global limit of 10 kB/s.

4.3 Smoothing

This naïve approach of delaying I/O operations tends to result in very bursty network behavior since we are blocking an I/O of any length for some time, and then letting it complete in full. As expected, this behavior is especially prevalent when operations are large. In the short term, burstiness may even result in *over shaping* as network conditions are changing, and the scheduler might have been able allocate more I/O to the stream in question. Figure 4 shows the extremity of this effect, where operations are large and rate limiting very aggressive.

The length of an I/O may also be unpredictable, especially in applications with network traffic driven by user input (e.g. interactive login sessions or games). Such applications are naturally bursty and it would be advantageous for Trickle to dampen these bursts.

Note that even if Trickle considered instantaneous bandwidth consumption in addition to the windowed average as netbrake[5] does, bursty behavior would still be present in many applications. When shaping is based on both instantaneous and average bandwidths, it is the hope that the buffers underneath the application layer will provide dampening. For I/Os (keep in mind that applications are allowed to make arbitrarily large I/O requests to the socket layer) with lengths approaching and exceeding the bandwidth $\times$ delay product, buffering provides little dampening.

Thus, we introduce techniques to *smooth* these bursts. The techniques we introduce are generic and apply equally to both instantaneous and TCP burstiness. Our technique makes use of two parameters to normalize traffic transmitted or received by the socket layer.

In the following discussion, we use the variable pointvalue to indicate the value of a point after scheduling, numpoints is the number of points allocated to

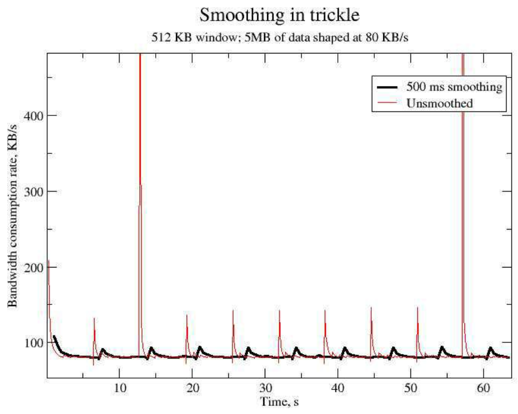


Figure 3: Measuring a windowed-average bandwidth consumption at the receiving end, this plot shows the effects of smoothing in Trickle.

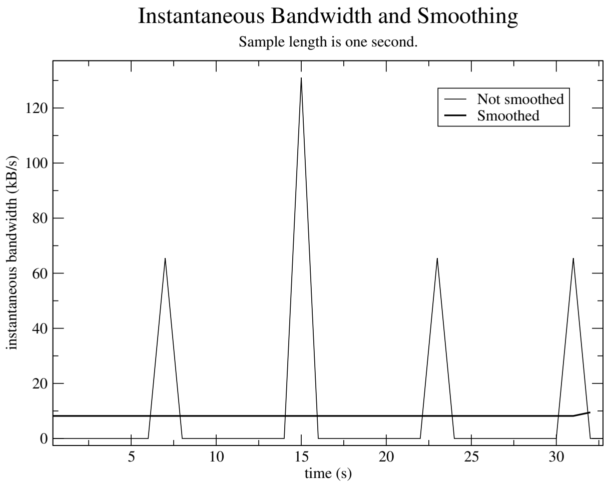


Figure 4: When dealing with large I/O operations, smoothing helps amortize bandwidth consumption.

the entity in question and length refers to the (original) length of the I/O being scheduled.

We first introduce a *time smoothing* parameter. We set the delay imposed on a socket to the minimum of the time smoothing parameter and the delay requested (by the process outlined in the previous subsection). If the time smoothing delay is the smaller of the two, the length is truncated so that the entity meets its adjusted rate allotment. This is called the *adjusted length*: $\mathsf{adjlen} = \mathsf{pointvalue} * \mathsf{numpoints} * \mathsf{timesmoothingparam}$. The purpose of the time smoothing parameter is to introduce a certain continuity in the data transfer.

We must be able to handle the case where the adjusted length is 0. That is, the time smoothing parameter is too small to send even one byte. To mitigate this situation, we introduce a *length smoothing* parameter. When the adjusted length is 0, we simply set the length to the length smoothing parameter, and adjust the delay accordingly: $\mathsf{delay} = \mathsf{length}/(\mathsf{pointvalue} * \mathsf{numpoints})$.

The effect of smoothing is illustrated in figure 3. Here, Iperf[27], a network measurement tool, was run for 60 seconds. The source node was rate limited with Trickle, which was configured to rate limit at 80 KB/s. The thin line indicates the resulting Iperf behavior without any smoothing applied. The thick line applies a time smoothing parameter of 500 ms. The transfer rates shown are instantaneous with a sampling rate once per second.

In practice, the scheduler deployed as follows: In a single Trickle instance, the entities are sockets, all with priority 1 and the global limit is either user specified or by `trickled`. `trickled` again uses the same scheduler: Here the entities are the instances of Trickle, and the global limit is specified by the user. Note that in this case, the scheduler does not need to apply delay or smoothing, it simply needs to report back to each instance of Trickle what its allotment is at that point in time.

4.4 Streams vs. Packets

One big difference between Trickle and in-kernel rate limiters is that Trickle only has access to much higher levels of the OSI layers. The only context available to Trickle are BSD sockets, while in-kernel rate limiters typically schedule discrete packets and reside in or around the network layer.

Trickle can only provide bandwidth shaping by delaying and truncating I/O on a socket (e.g. TCP/IP stream), and must rely on the underlying semantics in order to be effective. Furthermore, Trickle is affected by local buffering in every OSI[24] layer, and this effectively reduces the best reaction time Trickle can provide. Together, these conditions severely reduces the granularity at which Trickle can operate.

Since in-kernel rate limiters reside so low in the network stack, they may exercise complete control of the actual outgoing date rates. These rate limiters typically provide ingress rate limiting by a technique called “policing”. Policing amounts to simply dropping matching incoming packets even though there is no real local contention that would otherwise prevent them from being delivered. Note that policing is a different strategy for shaping incoming traffic. When a policing router drops a packet, it creates congestion in the view of the sending TCP as it will need to retransmit the lost packet(s). When detecting congestion, the sending TCP will reduce its congestion window, effectively throttling its rate of transmission. After shrinking its congestion window, the sending TCP has to expand its congestion window again by the slow start and congestion avoidance strategies. Trickle's approach avoids artificial congestion by shrinking the advertised TCP receiver window (`rwnd`), causing the sending TCP to artificially limit the amount of data it can send. One advantage of this technique is that policing makes TCP more volatile in a somewhat subtle way. In its steady-state, TCP is self-clocking, that is, it tries to inject a packet into the network for every packet that leaves the network. Thus, as networks get more congested, router queues fill up and round trip times (RTTs) increase, and thus the sending TCP slows its transmission rate. When policing, packets are dropped indiscriminately, and TCP has no chance to avoid the congestion by observing increasing RTTs. This results in a more volatile TCP state as the sending TCP will have to enter fast retransmit/recovery mode more frequently.

4.5 The Interactions of Delay, Smoothing and TCP

To recapitulate, Trickle shapes network traffic by delaying and truncating I/O requests according to a few simple parameters. Trickle attempts to reduce burstiness by smoothing, which in effect introduces some time and length normalization for the emitted I/O operations. We now explore how our shaping techniques interact with TCP.

For ingress traffic, this should result in a less volatile`rwnd` in the receiving TCP since the utilization of socket receive buffers have smaller variance.

Smoothing is also beneficial for the transmitting TCP. Because the data flow from the application layer is less bursty, the TCP does not have to deal with long idle times which may reduce responsiveness: It is standard practice to reduce the congestion window (`cwnd`) and perform slow start upon TCP idleness beyond one retransmission timeout (`RTO`)[20].

Smoothing may also be used for adapting to interactive network protocols. For example, a smaller time smoothing parameter should cause data to be sent in a more continuous and consistent manner, whereas the lack of smoothing would likely cause awkward pauses in user interactions.

Another tradeoff made when smoothing is that you are likely to loose some accuracy because of timing. When using timers in userland, the value used is the floor of the actual timeout you will get, and thus when sleeping on a timer just once for some I/O, the inaccuracy is amortized over the entirety of that I/O. However, smoothing is likely to break this I/O up into many smaller I/Os, and the timer inaccuracies may be more pronounced. The ultimate compromise here would be to use the effects of buffering whenever you can, and to use smoothing whenever you have to. This is left for future work.

5 Related Work

There are a number of generic rate limiting software solutions and one is included in nearly every major open source operating system. These operate in a mostly traditional manner (defining discrete packet queues and applying policies on these queues). What differentiates these utilities are what operating system(s) they run on and how expressive their policies are. Examples are AltQ[14], Netnice[21], Dummynet[23] and Netfilter[26].

Several network client and server applications incorporate rate limiting as a feature. For example, OpenSSH[8]'s has the ability to rate limit `scp` file transfers. `rsync`[9] too, features rate limiting. Both use a simple scheme that sleeps whenever the average or instantaneous bandwidth rates go beyond their threshold(s). `rsync` has the additional advantage that they may control the sender as well (their protocol is proprietary) and so bandwidth is shaped at the sender, which is easier and more sensible.

There are a few modules for Apache[1] which incorporate more advanced bandwidth shaping. These modules are typically application layer shapers: they exploit additional context within Apache and the HTTP protocol. For example, such modules could have the ability to perform rate limiting by particular cookies, or on CPU intensive CGI scripts.

Many peer-to-peer applications also offer traffic shaping. Again, here there is great opportunity to use application level knowledge to apply policies for shaping[2, 4].

Netbrake[5] is another bandwidth shaper that uses shared library preloading.[3] Like Trickle, it delays I/Os on sockets. Netbrake does not have the ability to coordinate bandwidth allocation amongst several instances. Netbrake calculates aggregate bandwidth usage for all sockets in a process, and shapes only according to this: That is, if a given socket I/O causes the global rate to exceed the specified limit, that socket is penalized with a delay as for bandwidth consumption to converge to the limit, and there no equivalent to the smoothing in Trickle. Thus, Netbrake does not retain a sense of "fairness" among sockets: One "overzealous" socket could cause delays in other sockets performing I/O at lower rates. This does not retain the TCP fairness semantics (nor does it attempt to), and could cause uneven application performance, one example being an application that uses different streams for control and data. Netbrake also does not distinguish between the two directions of data; incoming data will add to the same observed rate as outgoing data.

Netbrake is not semantically transparent (nor does it aim to be);

- it does not handle non-blocking I/O nor
- I/O multiplexing (`select()`, `poll()`, etc.) nor
- socket aliasing (`dup()`, etc.).

Trickle provides semantic transparency and the ability to provide fairness or managed differentiated bandwidth usage to different sockets or applications. Trickle also allows applications to cooperate as to retain (a) global bandwidth limit(s).

6 Future Work

Trickle does not have much control over how the lower layers of the network stack behave. A future area of exploration is to dynamically adjust any relevant socket options. Especially interesting is to adjust the socket send and receive buffers as to lessen the reaction time of Trickle's actions. Another area of future work is dynamic adjustment of smoothing settings, parameterized by various observed network characteristics and usage patterns (e.g. interactive, bulk transfers) of a particular socket.

There also exists a need for Trickle to employ more expressive and dynamic policies, for example adding the ability to shape by remote host or by protocol.

There are a few new and disparate interfaces for dealing with socket I/O multiplexing. In the BSD operating systems, there is the `kqueue`[18] event notification layer, Solaris has `/dev/poll`[13] and Linux `epoll`[19]. Trickle stands to gain from supporting these interfaces as they are becoming more pervasive.

By using a system call filter such as Systrace[22] or Ostia[17], Trickle could address its two highest impact issues. By using such a system call filter, Trickle could interposition itself in the system call layer, while still running entirely in userland, hence gaining the ability to work with statically linked binaries. Furthermore, these tools provide the means to actually enforce the usage of Trickle, thus enforcing bandwidth policies.

In order to do collaborative rate limiting when joining a new network, a user would have to manually find which host (if any) is running `trickled`. Trickle would thus benefit from some sort of service discovery protocol akin to DHCP[15]. Using Zeroconf[12] technologies could potentially prove beneficial.

7 Acknowledgments

The author would like to thank the following people for their sharp minds and eyes: Evan Cooke, Crispin Cowan (our shepherd), Kamran Kashef, Ken MacInnis, Joe McClain, Niels Provos (also for pushing and prodding to submit this paper), Andrew de los Reyes, Cynthia Wong, as well as the anonymous reviewers.

8 Availability

Trickle source code, documentation and other information is available under a BSD style license from

`http://monkey.org/~marius/trickle/`

9 Summary and Conclusion

Trickle provides a practical and portable solution to ad-hoc rate limiting which runs entirely in userland. It has been shown to work extremely well in practice, and none of its inherent limitations seem to be a problem for its target set of users.

Since the time Trickle was released in March, 2003, it has enjoyed a steady user base. It is widely used, especially by home users in need of ad-hoc rate limiting. Trickle has also been used in research.

Trickle works by interpositioning its middleware at the BSD socket abstraction layer which it can do this entirely in userland by preloading itself using the link editor present in Unix-like systems. Trickle has been reported to work on a wide variety of Unix-like operating systems including OpenBSD[7], NetBSD[6], FreeBSD[3], Linux[10] and Sun Solaris[11], and is by its very nature also architecture agnostic.

At the socket layer the number of ways an operation can be manipulated is limited. Furthermore, we gain no access to lower layers in the network stack at which rate limiters typically reside, and so we have to rely on the semantics of TCP to cause reductions in bandwidth consumption. We developed several techniques including *smoothing* that help normalize the behavior observed in the lower network layers and avoids bursty throughput.

There are many venues to explore in the future development of Trickle, and we believe it will remain a very useful utility for ad-hoc rate limiting. Furthermore, we expect that this is the typical usage case for rate limiting by end users requiring occasional service differentiation.

References

[1] Apache. `http://www.apache.org/`.

[2] Azureus - java bittorrent client. `http://azureus.sourceforge.net/`.

[3] FreeBSD: An advanced operating system. `http://www.freebsd.org/`.

[4] Kazaa 3.0. `http://www.kazaa.com/`.

[5] Netbrake. http://www.hping.org/netbrake/.

[6] NetBSD: the free, secure, and highly portable Unix-like Open Source operating system. http://www.netbsd.org/.

[7] OpenBSD: The proactively secure Unix-like operating system. http://www.openbsd.org/.

[8] OpenSSH. http://www.openssh.com/.

[9] Rsync. http://samba.anu.edu.au/rsync/.

[10] The Linux Kernel. http://www.kernel.org/.

[11] The Sun Solaris Operating System. http://wwws.sun.com/software/solaris/.

[12] The Zeroconf Working Group. http://www.zeroconf.org/.

[13] ACHARYA, S. Using the devpoll (/dev/poll) interface. http://access1.sun.com/techarticles/devpoll.html.

[14] CHO, K. Managing traffic with ALTQ. In *Proceedings of the USENIX 1999 Annual Technical Conference* (June 1999), pp. 121–128.

[15] DROMS, R. Dynamic Host Configuration Protocol. RFC 2131, 1997.

[16] FRALEIGH, C., MOON, S., LYLES, B., COTTON, C., KHAN, M., MOLL, D., ROCKELL, R., SEELY, T., AND DIOT, C. Packet-level traffic measurements from the sprint IP backbone. *IEEE Network* (2003).

[17] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium* (February 2004).

[18] LEMON, J. Kqueue - a generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), USENIX Association, pp. 141–153.

[19] LIBENZI, D. /dev/epoll home page. http://www.xmailserver.org/linux-patches/nio-improve.html.

[20] M. ALLMAN, V. PAXSON, W. S. TCP Congestion Control. RFC 2581, Apr 1999.

[21] OKUMURA, T., AND MOSSÉ, D. Virtualizing network i/o on end-host operating system: Operating system support for network control and resource protection. *IEEE Transactions on Computers* (2004).

[22] PROVOS, N. Improving host security with system call policies. In *Proceedings of the 11th USENIX Security Symposium* (Aug. 2003), USENIX, pp. 257–272.

[23] RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev. 27*, 1 (1997), 31–41.

[24] ROSE, M. T. *The Open Book: A Practical Perspective on OSI*. Prentice Hall, 1990.

[25] S. BOTTOMS, T. L., AND WASH, R. nlimit: A New Voluntary Bandwidth Limiting Facility. Unpublished manuscript, 2002.

[26] TEAM, T. N. C. The Netfilter Project: Packet Mangling for Linux 2.4. http://www.netfilter.org/, 1999.

[27] TIRUMALA, A. End-to-end bandwidth measurement using iperf. In *SC'2001 Conference CD* (Denver, Nov. 2001), ACM SIGARCH/IEEE. National Laboratory for Applied Network Research (NLANR).

Notes

[1]A small software component used to provide an interface to another software component (a trivial "adapter")

[2]Modern operating systems have introduced new and more efficient interfaces for multiplexing. These are discussed in section 6

[3]The author of Netbrake notes that Netbrake is intended to simply be a hack, and that it is not mature software.[5]

A Tool for Automated iptables Firewall Analysis

Robert Marmorstein
Department of Computer Science
The College of William & Mary
Williamsburg, VA 23185
rmmarm@CS.WM.EDU

Phil Kearns
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
kearns@CS.WM.EDU

Abstract

We describe `ITVal`, a tool that enables the efficient analysis of an iptables-based firewall. The underlying basis of `ITVal` is a library for the efficient manipulation of multi-way decision diagrams. We represent iptables rule sets and queries about the firewall defined by those rule sets as multi-way decision diagrams, and determine answers for the queries by manipulating the diagrams. In addition to discussing the design and implementation of ITVal, we describe how it can be used to detect and correct common firewall errors.

1 Introduction

As firewalls have become a cornerstone of many security policies, they have grown in power and in complexity. In addition to packet filtering, a typical firewall now provides stateful inspection and packet mangling. One side effect of these changes is the increasing difficulty of correctly configuring a firewall rule set. Subtle errors in a firewall configuration can be difficult to detect, but can open the doors to a malicious intruder or a denial of service attack.

Integrated into the Linux kernel is a firewall system called netfilter[16, 13, 4], which provides the internal hooks for the iptables packet filter. Both stateful inspection and advanced packet mangling are supported by iptables.

To configure iptables, a system administrator creates chains of filtering rules in which order is significant. This means that inserting rules in the wrong order can introduce errors. Additionally, because the number and size of the chains determines the complexity of the rule set, a firewall with many large chains can be very difficult to understand. Sometimes it is difficult even to know which requests for important network services are allowed to pass through the firewall filter.

Configuration errors can be extremely difficult to identify in a real world system. Figure 1 shows a firewall that secures an internal network 192.168.2.0/24 from intrusions by hosts on an unsecured wireless network 192.168.1.0/24. All traffic, including HTTP traffic, should be dropped from that insecure network. Rule 1 drops any incoming ICMP packets. Rule 2 drops traffic from the insecure network. The remaining rules secure various services and let in traffic to the web server. All other traffic is dropped, unless it comes from a secure subnet 113.192.10.0/24.

Let's say the administrator decides to modify this configuration to allow IPP printing traffic (port 631) from trusted machines to pass through the firewall to the secure network. If she inserts an accept rule in the wrong place, she can produce the incorrect configuration in figure 2. This configuration allows printing service from the insecure network, because the new rule has been inserted before the rule which restricts the insecure subnet. Switching rules 2 and 3 yields a correct configuration. This sort of error becomes harder to detect as the number of rules grows and the complexity of their structure increases.

Consider also the firewall rule set described in figure 3, which protects an internal subnet 192.168.2.0/24 from the outside world. Can mail be forwarded through this filter from an arbitrary host using SMTP? The answer is yes. At first glance, it appears that only hosts from 192.168.2.0/24 can access SMTP (they are granted access in rule 5). Rule 6, however, gives access to any host on any port, provided it is part of an established connection. In order for SMTP traffic to pass from an outside host through the firewall, one of the machines in the 192.168.2.0/24 subnet must establish an SMTP connection to the unauthorized host. Once a connection has been established (via TCP handshake), rule 6 will allow arbitrary access. This scenario could happen if an internal machine is compromised by a virus or Trojan. It could then open up a mail server, connect to the external host, and forward spam to anyone on the internal network.

Chain FORWARD (policy DROP)						
	target	**prot**	**opt**	**source**	**destination**	**flags**
1	DROP	ICMP	–	anywhere	192.168.2.0/24	
2	DROP	all	–	192.168.1.0/24	192.168.2.0/24	
3	DROP	TCP	–	anywhere	192.168.2.0/24	TCP dpt:domain
4	ACCEPT	TCP	–	anywhere	192.168.2.0/24	TCP dpt:HTTP
5	DROP	TCP	–	anywhere	192.168.2.0/24	TCP dpt:RSH
6	ACCEPT	all	–	113.192.10.0/24	192.168.2.0/24	

Figure 1: A sample firewall that secures subnet 192.168.2.0/24 against intrustions from untrusted network 192.168.1.0/24

Chain FORWARD (policy DROP)						
	target	**prot**	**opt**	**source**	**destination**	**flags**
1	DROP	ICMP	–	anywhere	192.168.2.0/24	
2	ACCEPT	all	–	anywhere	192.168.2.0/24	TCP dpt:631
3	DROP	all	–	192.168.1.0/24	192.168.2.0/24	
4	DROP	TCP	–	anywhere	192.168.2.0/24	TCP dpt:domain
5	ACCEPT	TCP	–	anywhere	192.168.2.0/24	TCP dpt:HTTP
6	DROP	TCP	–	anywhere	192.168.2.0/24	TCP dpt:RSH
7	ACCEPT	all	–	113.192.10.0/24	192.168.2.0/24	

Figure 2: A misconfigured firewall that allows the untrusted network to access printing services

While these errors can easily be avoided by a careful system administrator, far more subtle and complicated errors can evolve as a firewall rule set grows and is modified to permit new services or patch new security vulnerabilities.

In a survey of 37 corporate firewalls, Wool[18] discovered an average of 7 configuration errors per system. While his study did not examine iptables systems, it is not unreasonable to assume that the error rate for Linux systems is comparable to those of Checkpoint firewalls.

In [2], Alexander points out that because firewalls are hard to configure, they often fail to prevent spoofing attacks from one internal subnet to another, which can compromise vital financial and planning information.

1.1 Existing Tools

There have been several different attempts to address the problem of firewall misconfiguration. These solution can be broken into two basic categories: active testing and passive testing. Active testing uses tools such as SATAN[9], nessus[3], or Ftester[5] to subject a firewall to a sequence of carefully crafted packets and see which ones get through. Passive testing involves an offline analysis of the firewall configuration.

Since it is impossible to test every possible packet, active tools test only a portion of the firewall configuration. This makes them well-suited for detecting specific vulnerabilities and for detecting implementation bugs in the firewall software, but not for generating trust in the overall security of a firewall configuration. It also means that testing can interfere with normal network activity.

The current state of the art in passive analysis is a commercial tool produced by Algorithmic Security called "Algosec Firewall Analyzer", which is available for PIX and Checkpoint FW-1 firewalls. It is a closed-source commercial project based on Wool's Fang[1] and Lumeta[17] engines. Fang allowed the user to perform simple queries such as "what packets can reach the mail server." In Lumeta, the developers replaced Fang's query functionality with a graphical tool that checks for specific configuration errors. Algosec is a more powerful commercial version of Lumeta. Each of their systems is capable of analyzing multiple firewalls in a specified network topology.

Another branch of research has focused on simplifying a firewall configuration by removing redundant and conflicting rules. In [10], Gouda and Liu present an algorithm for constructing a *firewall decision diagram* and applying reduction techniques to derive a complete, compact, and consistent firewall. Their technique can reduce the complexity of a poorly configured firewall and uncover some configuration errors, but has a different purpose than such engines as Algosec and SATAN. Gouda and Liu's work focuses on errors in the structure of a firewall rule set rather than its substance.

Chain FORWARD (policy DROP)						
	target	**prot**	**opt**	**source**	**destination**	**flags**
1	ACCEPT	ICMP	–	anywhere	192.168.2.0/24	
2	ACCEPT	TCP	–	anywhere	192.168.2.0/24	TCP dpt:ssh flags:SYN,ACK/SYN
3	ACCEPT	UDP	–	anywhere	192.168.2.0/24	UDP dpt:domain
4	ACCEPT	all	–	113.117.1.4	192.168.2.0/24	
5	ACCEPT	all	–	192.168.2.0/24	anywhere	
6	ACCEPT	all	–	anywhere	anywhere	state RELATED,ESTABLISHED

Figure 3: A stateful ruleset which allows SMTP access for established connections.

None of these techniques is widely available in an open source tool which can be used with iptables.

An interesting intermediary between active and passive tools is Russell's netfilter simulator[14]. The simulator is intended to be used for debugging kernel hooks in netfilter, and provides very low-level access to the internals of netfilter, so it is not by itself suitable as a query tool for non-developers, but could perhaps be used as the basis of a more general query library.

2 `ITVal`, An Open Source Tool

In this paper, we present `ITVal`, an open-source firewall analysis tool for iptables. The tool takes as inputs a firewall rule set as generated by the output of the "`iptables -L -n`" command and a query file written in a simple language we describe in section 3. It then calculates the set of packets which can be accepted by the firewall and produces the answers to each query in the query file.

In the query file, the user can ask questions such as "What services can be reached on host X?" or "Which machines can be reached with SSH?" The analysis engine resolves the queries using a very efficient decision diagram data structure and prints the results on standard output. The analysis engine can handle many of the features of iptables, including stateful inspection.

Using this tool, the system administrator can check important security properties before and after making a change to the firewall. If the query engine returns an unexpected result, he can examine his changes for errors and reapply the security check.

The analysis engine is implemented using FDDL[12], a Multi-way Decision Diagram (MDD) library. We chose to use MDDs[8] over Binary Decision Diagrams(BDDs)[6] because they are better suited for representing integral values such as ports and IP address.

Decision diagrams have been previously used to represent firewall rules. In [15], Hazelhurst et al. showed that firewalls could be represented using Binary Decision Diagrams and implemented a limited number of queries using formal logic. In [11], they shifted their focus toward improving firewall performance by representing the rule set using decision diagrams. Christiansen and Fleury took this a step further in [7] by implementing an Interval Decision Diagram based packet filter for use with netfilter.

These projects, and that of Gouda cited above, have a different motivation from that of `ITVal`. While these projects sought to enhance performance and provide a formal characterization of firewall rule sets, our aim is to provide a simple, plain English query language that simplifies accurate firewall configuration.

3 Query Language

The analysis tool provides a straightforward query language which allows complex queries to be built from simple primitives. An example query file is shown in figure 4. The query file consists of a set of group and service definitions followed by one or more query statements. The first four lines of figure 4 are definitions. The next four lines are query statements.

3.1 Query Statements

Query statements begin with the word QUERY followed by a *subject*, an optional *input chain*, a *condition*, and a semicolon. The subject of the query specifies what information should be printed about packets that match the query. For instance, in line 5, the subject "DADDY" indicates that the destination address should be printed. The valid subjects are:

- SADDY : Source Address
- DADDY : Destination Address
- SPORT : Source Port
- DPORT : Destination Port
- STATE : Connection State

```
1   GROUP internalnet 68.10.120.* 68.10.121.*;
2   GROUP wlan 68.10.122.*;

3   SERVICE mail TCP 25 TCP 110;
4   SERVICE ftp TCP 21 TCP 20;

5   QUERY DADDY FROM wlan AND (FOR mail OR FOR TCP 80);
6   QUERY SPORT OUTPUT TO internalnet AND FOR ftp AND IN NEW;

7   QUERY SADDY TO internalnet AND FOR 68.11.230.45 AND
    (NOT IN NEW AND NOT IN RELATED);

8   QUERY DPORT FROM internalnet AND TO wlan AND
    (IN NEW OR IN ESTABLISHED);
```

Figure 4: An example query file

A query statement can optionally contain the name of an input chain to use. The input chain must be one of the three built-in chains: INPUT, FORWARD, or OUTPUT. If no input chain is explicitly given, the analysis engine assumes that the FORWARD chain should be considered. Line 6 of the example specifies that the OUTPUT chain should be considered rather than the FORWARD chain. The rest of the query statement consists of a condition which specifies the packets to consider.

3.2 Simple Conditions

The query engine allows the user to build complex conditions out of very simple conditions. Conditions are built from seven simple primitives:

- FROM <address group> : Specifies one or more source addresses to match.
- TO <address group> : Specifies one or more destination addresses to match.
- ON <service> : Specifies one or more source ports to match.
- FOR <service> : Specifies one or more destination ports to match.
- WITH <flag> : Specifies TCP flags to match against.
- IN <state> : Specifies a connection state to match.
- LOGGED : indicates that exists a rule potentially logging the arrival of the packet

Each of the primitives selects those packets that are accepted and that match the specified criteria. For instance "FROM 127.0.0.1" specifies those packets accepted by the firewall which are outbound from localhost.

For the FROM and TO queries, the address group can either be the name of a predefined address group or the numeric IP address of a host. Asterisks may be used in numerical addresses to describe an entire subnet at once.

For the ON and FOR queries, the service can be either the name of a predefined service or the numeric port number of the service preceded by the protocol type. An asterisk can be used to match all packets of the given protocol type. The protocol type can either be TCP, UDP, BOTH, or ICMP. If ICMP is chosen, the ICMP packet type number should be specified instead of a numerical port. If BOTH is specified, the analysis engine will match both TCP and UDP packets.

For the WITH primitive, the recognized TCP flags are URG, PSH, RST, FIN, SYN, and ACK.

For the IN primitive, the connection state can be either INVALID, NEW, ESTABLISHED, or RELATED.

The LOGGED primitive stands on its own without any parameters. It indicates that a packet may have been logged by the firewall. Since iptables LOG rules can specify time-related and other external criteria for logging, there is no guarantee that every matching packet will actually be logged.

3.3 Complex Queries

The boolean connectives NOT, AND, and OR allow the user to posit queries of arbitrary complexity. These operators work as one would expect. "NOT FROM TCP 21" matches against all accepted packets which are not TCP packets on port 21. "FOR mail OR FROM 127.0.0.1" selects both mail packets and packets outbound from localhost. Parentheses may be used to disambiguate subexpressions containing multiple operators.

3.4 Group and Service definitions

If the user had to explicitly mention every host address explicitly in every query, creating a query file would be a tedious and error prone process. To address this issue, we allow named groups of addresses to be defined and used throughout the query file. The syntax for specifying a group is the word GROUP followed by a name and a space separated list of addresses. Asterisks may be used to include entire subnets at once. Group names must consist entirely of letters and may not match any keyword of the query language.

Similarly, named groups of services may be defined. The syntax for defining a service is the word SERVICE followed by a name and a space separated list of protocols and ports.

4 Implementation

When invoked on an iptables rule set as given in the output of

```
iptables -n -L
```

and a query file, the `ITVal` analysis engine parses the rule set and builds an MDD representing the set of packets accepted by the firewall for each of the built-in chains. Then it parses the query file and generates an MDD representing the set of packets which match the condition of each query. Using an efficient MDD intersection operator, it calculates the set of packets that are both accepted by the firewall and match the condition of the query. Then it displays the information specified in the subject of the query.

4.1 Using Decision Diagrams

A multi-way decision diagram(MDD) is a directed acyclic graph in which the nodes are organized into $K+1$ levels and all arcs from a node at non-terminal level $k > 0$ point to nodes at level $k-1$. In this application, every path through the MDD represents a packet potentially received by the firewall. Each of the non-terminal levels of the MDD corresponds to a specific attribute of the packet. For instance, in figure 5, the MDD corresponding to the rule set of figure 1, level 20 represents the first octet of the source address.

Level 0 is a special *terminal* level which, for rule set MDDs, represents the target of the firewall rule (ACCEPT, DROP, LOG, or a user-defined chain) as a unique integer index. We also reserve terminal index 0 to mean "not yet specified." For the query MDDs, nodes at the terminal level express whether or not the packet matches the query criteria. Terminal node 0 represents "does not match" and terminal node 1 represents "matches".

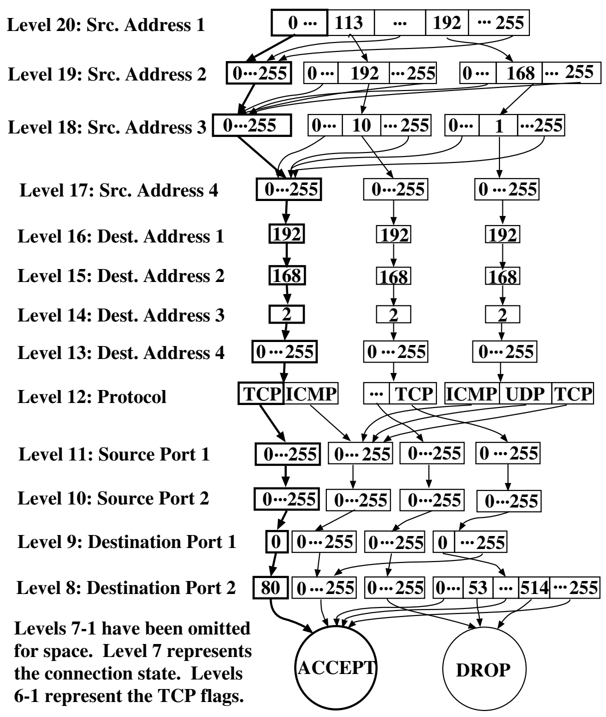


Figure 5: MDD for the rule set in figure 1

A non-terminal node at level k represents a subset of packets that share some attributes. An arc from a node at level k to a node at level $k-1$ represents a choice of value for the attribute represented at level k.

When many arcs from a node point to the same child, we use ellipses in the figure to save space. In the actual MDD there would be arcs for each value we have hidden in this manner.

Nodes at each level are stored in a dynamic array and are referenced by a unique integer index. At every level, we reserve index 0 for a special node, node zero, which represents the empty set. This can be thought of as a node with all its arcs pointing to node zero at the level below. To save a small amount of memory, we do not explicitly store node zero. We denote node n of level k as $<k, n>$ and the i^{th} arc of that node as $<k, n>[i]$.

To see that an HTTP packet from 68.10.1.3 to 192.168.2.10 is accepted by the firewall, start with the node at level 20 of the MDD. Since the first source octet of the packet is 68, which falls between 0 and 113, follow the first arc to the highlighted node at level 19. Now there is only one arc to follow. Since 10 falls between 0 and 255, follow the highlighted arc to the next level. Again, 1 falls between 0 and 255 so follow the arc to level 17. The last octet of the source address is 3, which falls between 0 and 255, so follow the highlighted arc to the node at level 16.

```
node_index MakeMDDFromRule(ParsedRule pr)
1  old = LookUpTarget($pr.target$).
2  for $k = 1$ to $K$:
3    node $n =$ NewNode($k$).
4    for $i$ = 0 to MaxValue($k$):
5      if $i < pr.low[k]$ and $i > pr.high[k]$:
6        $<k, n>$[i]=$old$.
7    $old$ =CheckForDuplicates($n$).
8  return $n$.
```

Figure 6: Algorithm for building an MDD from a rule

Level 16 represents the first octet of the destination address, which for our example is 192. Since there is an arc for 192, proceed to level 15. If the destination address had been 193.1.1.1, you would know that the packet is dropped by the firewall, since there is no arc for 193 and DROP is the default policy. Instead, at level 15, examine the second octet of the destination address. Since there is an arc for 168, proceed to level 14. Continue in this manner to level 12.

At level 12, there is an arc for TCP and an arc for ICMP. Since HTTP is a TCP protocol, follow the arc for TCP to the highlighted node at level 11. Continue in this manner until you reach the node at terminal level 0. Since it is the ACCEPT node, the packet will be accepted by the firewall.

4.2 Building an MDD for a Filter Rule

In order to construct an MDD for a rule, we first parse the rule into target, source address, destination address, source port, destination port, protocol, state, and flag components. From these components, we create a parsedrule, which represents each component as an integer. We define an operation MakeMDDFromRule, show in figure 6, which takes the parsed firewall rule and returns the root node $<K, n>$ of an MDD representing that rule.

The algorithm starts at level 0 and builds upward toward the root node. At each level, it creates new nodes that represent the criteria of the parsed rule. In line 1, Node $<0, n>$ is determined directly by finding the equivalent integer index of the rule target. For ACCEPT, DROP, and LOG targets this is a predefined constant less than 4. For user-defined rules, the index comes from a pre-generated table that maps the user-defined chains, in the order of their discovery during parsing, to integers greater than 3.

Lines $2 - 7$ construct nodes at levels 1 through K. The call to NewNode in line 3 creates a new node and initializes all its arcs to point at node zero. Lines $4 - 7$ examine each potential value i of filter rule attribute k. If i falls within the range specified by the parsed rule, arc $<k, n>[i]$ is connected to node $<k-1, old>$. Otherwise, the arc is left at its default value, which points to node zero.

In line 7, we have considered all the potential values of attribute k, so we now call CheckForDuplicates, which uses hashing to identify any nodes that exactly duplicate node $<k, n>$. If such a node exists, $<k, n>$ is freed and CheckForDuplicates returns the index of the duplicate node. Otherwise, it returns $<k, n>$.

```
mdd ConvertChain(rule* tup, mdd inMDD)
1   if $tup$ ==NULL:
2     return inMDD.
3   inMDD = ConvertChain(tup.next, inMDD).
4   interMDD = MakeMDDFromRule(tup).
5   if target(tup) == DROP or target(tup) == ACCEPT:
6     return Replace(K, inMdd, interMDD).
7   otherwise:
8     chain = LookUpChain(target(tup)).
9     interMDD = ConvertChain(chain.first_rule, interMDD).
10    interMDD = FilterTerminals(interMDD).
11    return Replace(K, inMDD, interMDD).
```

Figure 7: Algorithm for Constructing a Chain MDD

4.3 Converting chains to MDDs

In order to construct the MDD for an entire rule set, we consider chains using the algorithm shown in figure 7. To simplify this discussion, we omit the handling of LOG rules from the algorithm. LOG rules are implemented by maintaining an additional MDD for each chain which describes the set of logged packets.

The algorithm, ConvertChain, takes as inputs a chain, represented as a linked list of parsed rules, and an MDD storing the set of packets accepted by the rules seen so far. It recursively traverses the chain one rule at a time (in reverse order) to build an MDD representing the entire chain.

We initially call ConvertChain by passing in the first rule of the parsed FORWARD, INPUT, or OUTPUT chain and an MDD describing the default policy. This initial MDD consists of a single node, $<k, n>$, at each level. All arcs of the node at level 1 point to the index of the default target. All arcs of the nodes at levels $k > 1$ point to node $<k-1, n>$.

Lines 1 through 3 of ConvertChain place the rules of the chain on the call stack so that the rules can be processed in reverse order. The algorithm traverses the linked list until it reaches the end and then returns so that tup points to the last rule of the chain.

Line 4 creates an MDD representation of the rule as described in section 4.2. If the rule is a simple ACCEPT or DROP rule, line 6 uses a Replace operator to mask the

```
mdd Replace(level $k$, node_index $p$, node_index $q$)
1   if $p$==0 then return $p$.
2   if $q$==0 then return $p$.
3   if $p$.level == 0 then return $q$.
4   if $r$ =ReplaceCache[$k$,$p$,$q$] != $-1$ then return $r$.
5   $r$ =NewNode(k).
6   for $i = 0$ to MaxValue($k$):
7     $u$=Replace($k - 1$, <$k$, $p$>[i], <$k$, $q$>[i]).
8     <$k$, $r$>[$i$]=$u$.
9   $r$ = CheckForDuplicates(<$k$, $r$>).
10  ReplaceCache[$k$,$p$,$q$] = $r$.
11  return $r$.
```

Figure 8: MDD Replace Operation

new rule over any existing rules in the chain. Pseudocode for Replace is shown in figure 8.

Replace descends the MDD recursively, starting from the root node. When it reaches a terminal node, it returns node $<k, q>$ if $<k, p>$ and $<k, q>$ are both non-zero. Otherwise, it returns $<k, p>$.

Calling Replace on $inMDD$ and $interMDD$ produces a new MDD in which every rule of $interMDD$ masks any matching rule in $inMDD$.

Lines $8 - 11$ of ConvertChain handle the case where the target of the new rule is a user-defined chain. In line 9, an MDD representing the criteria portion of the rule is created using a recursive call to ConvertChain. Because iptables does not allow cyclic references in chain targets, the recursion is guaranteed to terminate.

When we return from the recursion, the ACCEPT and DROP rules of the user-defined chain will have been applied to our intermediate MDD, but some paths through the MDD may still lead to the chain-name target. These paths represent packets that are unaltered by the chain. We mask these out in line 10, and use the Replace operator to mask the new MDD over the result.

The MDD created by ConvertChain must only affect packets that match the criteria for the rule under consideration. The target chain, however, may have more general rules that affect packets that do not match those criteria. To avoid this problem, the replace operator only modifies packets that do not map to the "unspecified" terminal, terminal 0. Since the default policy matches every packet to some terminal, this will always be the case when applying ACCEPT and DROP rules to the top-level chain. When applying rules to a target chain, however, rules outside the criteria under consideration will map to 0 and be ignored by the replace operator.

4.4 Resolving Queries

The MDD for a query condition is constructed by joining smaller MDDs together according to the structure of the query. The MDD for a primitive, such as FROM, is

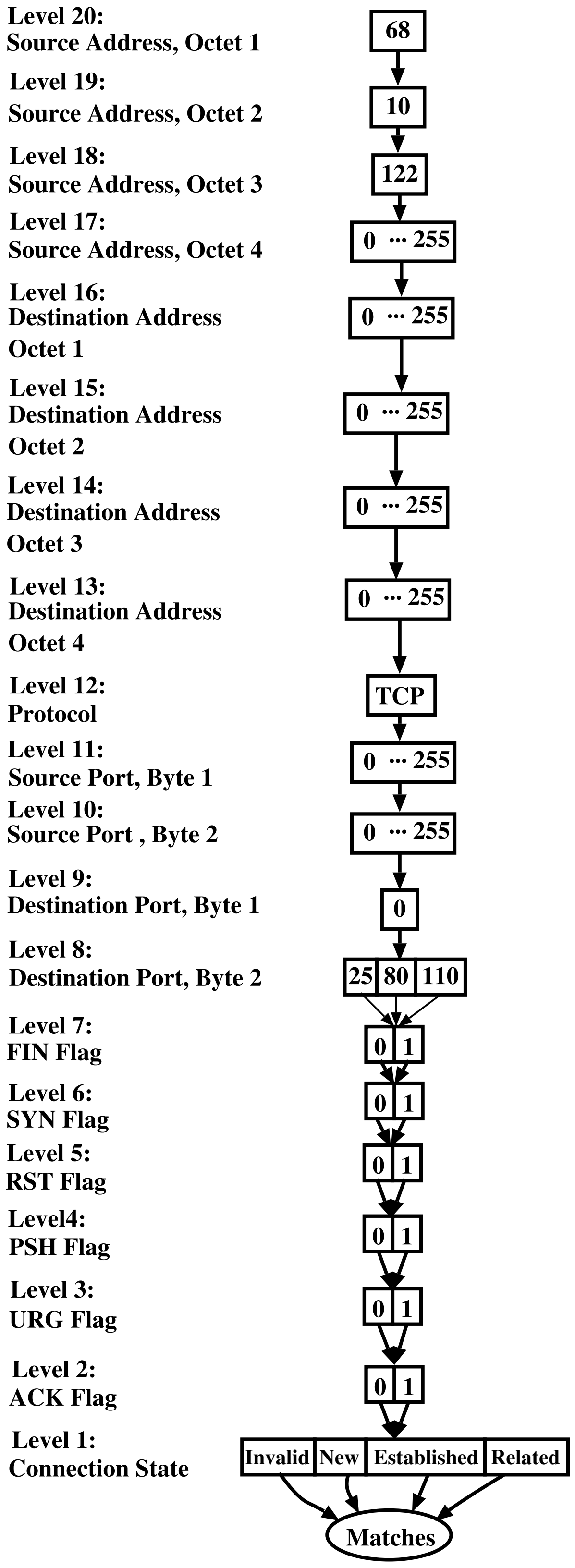

Level 20: Source Address, Octet 1
68
Level 19: Source Address, Octet 2
10
Level 18: Source Address, Octet 3
122
Level 17: Source Address, Octet 4
0 ··· 255
Level 16: Destination Address Octet 1
0 ··· 255
Level 15: Destination Address Octet 2
0 ··· 255
Level 14: Destination Address Octet 3
0 ··· 255
Level 13: Destination Address Octet 4
0 ··· 255
Level 12: Protocol
TCP
Level 11: Source Port, Byte 1
0 ··· 255
Level 10: Source Port , Byte 2
0 ··· 255
Level 9: Destination Port, Byte 1
0
Level 8: Destination Port, Byte 2
25 80 110
Level 7: FIN Flag
0 1
Level 6: SYN Flag
0 1
Level 5: RST Flag
0 1
Level4: PSH Flag
0 1
Level 3: URG Flag
0 1
Level 2: ACK Flag
0 1
Level 1: Connection State
Invalid New Established Related
Matches

Figure 9: MDD for the query on line 5 of figure 4

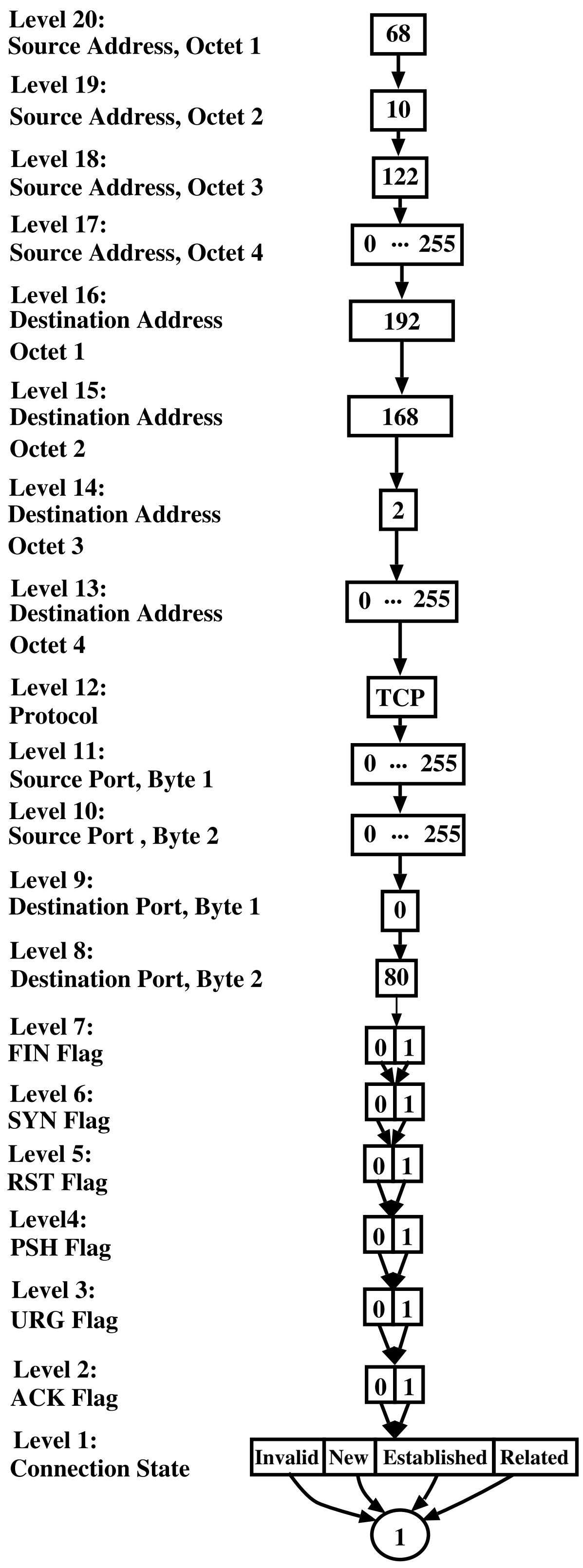


Figure 10: The result MDD for the running example

```
node IntersectMDD(level k, node p, node q)
1  if $k$ == 0:
2    if $p$==ACCEPT and $q$==1 then return 1.
3    else return 0.
```

Figure 11: Modifications to Intersection Operator

```
node IntersectMDD(level k, node p, node q)
1   if $k$ == 0:
2     if $p$==1 and $q$==1 then return 1.
3     else return 0.
4   if $p$ == 0 or $q$ == 0 then return 0.
5   if ($k$,$p$,$q$) is in the cache, return the cached result.
6   $result$ = NewMDDNode().
7   for $value$ from 0 to MaxValue($k$):
8     result[$value$] =
        IntersectMDD(k-1, $p[value]$, $q[value]$).
9   result = CheckForDuplicates(<$k$, $result$>).
10  store ($k$,$p$,$q$)=result in the cache
11  return $result$.
```

Figure 12: MDD Intersection Algorithm

built in the same manner as the MDD for a rule which matches every packet specified by the operator. Instead of ACCEPT, however, the terminal node is “matches”.

The primitive MDDs are then joined using an MDD union and the MDD intersection operator in figure 12 to produce the final query condition. The union operator is derived directly from the intersection operator by modifying the base cases.

The function $\mathrm{MaxValue}$ returns the maximum value for the field associated with level k. For instance, the maximum value of level 20 is 255, since level 20 represents the first octet of the source IP address. A cache is used to improve performance. Without the cache we might need to compute the intersection of a pair of nodes several times. Using the cache, we are guaranteed to compute it only once.

In order to resolve a query, we use FDDL to compute the intersection of the query MDD and the rule set MDD. The intersection algorithm for combining a query MDD and a rule MDD differs only slightly from the intersection operator used to combine query MDDs. The modified intersection algorithm replaces lines $1 - 3$ of figure 12 with those in figure 11 and produces an MDD which represents those packets that both satisfy the query and are accepted by the iptables rule set.

The intersection operation is linear in the product of the number of nodes in each MDD. Since the number of nodes can be exponentially smaller than the number of possible packets, queries can be performed very rapidly.

The result of intersecting the MDDs in our running

```
GROUP internalnet 68.10.120.* 68.10.121.*
GROUP wlan 68.10.122.8

SERVICE mail TCP 25 TCP 110
SERVICE ftp TCP 21 TCP 20

QUERY DADDY FROM wlan AND
                 (FOR mail OR FOR TCP 80)
Addresses: 192.168.2.*
256 results.
```

Figure 13: ITVal output for sample query

1	GROUP wlan 192.168.1.*;
2	SERVICE special ICMP * TCP 53 TCP 80 TCP 222;
3	QUERY DPORT FROM wlan;
4	QUERY DADDY FOR ICMP *;
5	QUERY SADDY FOR TCP 53;
6	QUERY SADDY FOR TCP 80;
7	QUERY SADDY FOR TCP 222;
8	QUERY SADDY NOT (FOR special OR FROM wlan);

Figure 14: Assertions for a hypothetical firewall

example is shown in figure 10. To see which packets are represented by the MDD, we start with the node at level 20. This node has a single arc representing the value 68, so all packets in the result have a source address that begins with 68.

Following the arc, we reach another node with a single arc. This node represents all packets with second source octet equal to 10. Continuing in this manner, we realize that the source address of all packets in the result must be in the group 68.10.122.* and the destination address must be in the group 192.168.2.*. The protocol must be TCP and the source port can have any value, but the destination port must be port 80, the HTTP port. When we continue down the graph, we realize that the result contains packets with any TCP flag condition and in any connection state. In other words, the result of our query is exactly: all HTTP packets from 68.10.122.* to 192.168.2.*.

In the context of `ITVal`, the output of the sample query applied to the sample rule set is shown in figure 13. Note that the human-readable output corresponds directly to the output MDD of figure 10.

5 Using ITVal

To illustrate how a hypothetical system administrator might use ITVal to detect and correct configuration errors, we return to the rule set described in figure 1. The query file shown in figure 14 can be used to verify all the

```
GROUP wlan 192.168.1.*;
SERVICE special ICMP * TCP 53 TCP 80
                TCP 222;
QUERY DPORT FROM wlan;
Ports:
0 results.
QUERY DADDY FOR ICMP *;
Addresses:
0 results.
QUERY SADDY FOR TCP 53;
Addresses:
0 results.
QUERY SADDY FOR TCP 80;
Addresses: [0-191].*.*.*
           192.[0-167].*.*
           192.168.0.*
           192.168.[2-255].*
           192.[169-255].*.*
           [193-255].*.*.*
4278190080 results.
QUERY SADDY FOR TCP 222;
Addresses:
0 results.
QUERY SADDY NOT (FOR special OR
            FROM wlan);
Addresses: 113.192.10.*
256 results.
```

Figure 15: Output of ITVal on the initial rule set from Fig. 1

important assertions about this network. For instance, the first query lists all services which can be accessed by the wireless network. The result of this query should be the empty set, since we want to restrict all access from that network. Similarly, the last query lists all hosts not on the wireless network that can access a service other than those explicitly permitted or denied. Only hosts from the trusted 113.192.10.0/24 network should appear in the answer to this query.

Running ITVal on the initial configuration gives the output shown in figure 15. In order to save space, we have grouped the output of the fourth query into ranges. The actual output would explicitly list each distinguishable subnet. It is easy to verify that all the requirements are satisfied. Suppose the administrator makes the incorrect change shown in figure 2. Running ITVal on this new rule set produces figure 16.

Because the first query no longer produces an empty result, it is evident that this rule set is incorrect. Realizing her mistake, the system administrator can then move the rule to its correct location in the rule set. Now the output of the first query will once again be the empty set.

```
GROUP wlan 192.168.1.*;
SERVICE special ICMP * TCP 53 TCP 80
                TCP 222;
QUERY DPORT FROM wlan;
Ports: 631
1 result.
QUERY DADDY FOR ICMP *;
Addresses:
0 results.
QUERY SADDY FOR TCP 53;
Addresses:
0 results.
QUERY SADDY FOR TCP 80;
Addresses: [0-191].*.*.*
           192.[0-167].*.*
           192.168.0.*
           192.168.[2-255].*
           192.[169-255].*.*
           [193-255].*.*.*
4278190080 results.
QUERY SADDY FOR TCP 222;
Addresses:
0 results.
QUERY SADDY NOT (FOR special OR
            FROM wlan);
Addresses: *.*.*.*
4294967296 results.
```

Figure 16: Output of ITVal after an incorrect modification on the rule set from Fig. 2

6 Conclusion

Using ITVal, a system administrator can quickly and easily verify that their firewall system important security requirements. One advantage of the query language is that generic queries generated for one firewall system can be employed on another firewall system with few modifications. This means that even without a complete understanding of the query language syntax, a system administrator can use ITVal to check fundamental security properties. Furthermore, queries are easier to generate correctly than the firewall rule sets, because they are more general, not order dependent, and don't depend on a complex interaction between independent chains.

There are a few areas in which ITVal needs further development, however. First, several types of packet mangling, such as masquerading and NAT, should be supported. Second, to enhance the applicability of the tool to real systems, it needs a mechanism for composition of firewalls in an arbitrary topology. Third, the tool needs a better output mechanism that can display results more concisely and allow the user to specify query subjects containing multiple fields.

Currently, queries that generate a large number of results can be difficult to interpret. A better interface could hide or group irrelevant data so that critical information stands out more clearly. In the long term, a graphical or interactive interface for displaying output would be a logical step.

The tool, as currently described, is available at `http://www.cs.wm.edu/˜rmmarm/ITVal/`.

References

[1] Mayer. Avishai Wool Alain and Elisha Ziskind. Fang: A firewall analysis engine. In *Proceedings of the IEEE Symposium on Security and Privacy*, MAY 2000.

[2] Steven Alexander. The importance of securing workstations. *;login:*, 30(1):23–26, February 2005.

[3] Harry Anderson. *Introduction to Nessus*, October 2003.

[4] Oskar Andreasson. *Iptables tutorial 1.1.19*, 2001.

[5] Andrea Barisani. Testing firewalls and ids with ftester. In *TISC Insight*, volume 5, 2001.

[6] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] Mikkel Christiansen and Emmanuel Fleury. Improving firewalls using bric(k)s. *BRICS Newsletter*, 11:56–59, December 2001.

[8] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. *Lecture Notes in Computer Science*, 1825:103–??, 2000.

[9] Dan Farmer and Wietse Venema. *SATAN: Security Administrator's Tool for Analyzing Networks*, 1995.

[10] Mohamed G. Gouda and Alex X. Liu. Firewall design: Consistency, completeness, and compactness. In *Proceedings of the International Conference on Distributed Computing Systems*. IEEE Computer Society, March 2004.

[11] Scott Hazelhurst. A proposal for dynamic access lists for tcp/ip packet filtering. Technical Report TR-Wits-CS-2001-2, University of Witwatersrand, April 2001.

[12] Robert Marmorstein. Designing and implementing a user library for manipulation of multiway decision diagrams. MS Project Report, Department of Computer Science,

The College of William and Mary, 2004. `http://www.cs.wm.edu/~rmmarm/Pubs/710paper.pdf`.

[13] Rusty Russel. *Linux 2.4 Packet Filtering HOWTO*, 2002.

[14] Rusty Russell and Jeremy Kerr. *Netfilter Simulation Environment*, 2004.

[15] Anton Fatti Scott Hazelhurst and Andrew Henwood. Binary decision diagram representation of firewall and router access lists. Technical Report TR-Wits-CS-1998-3, University of Witwatersrand, October 1998.

[16] Harald Welte. *netfilter/iptables FAQ*, 2003.

[17] Avishai Wool. Architecting the lumeta firewall analyzer. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[18] Avishai Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, June 2004.

Grave Robbers from Outer Space
Using 9P2000 Under Linux

Eric Van Hensbergen
IBM Austin Research Lab
Ron Minnich
Los Alamos National Labs

Abstract

This paper describes the implementation and use of the Plan 9 distributed resource protocol 9P under the Linux 2.6 operating system. The use of the 9P protocol along with the recent addition of private name spaces to the 2.6 kernel creates a foundation for seamless distributed computing using Linux. We review the design principles and benefits of Plan 9 distributed systems, go over the basics of the 9P protocol, describe 9P extensions to better support UNIX® file systems, and show some example Linux distributed applications using 9P to provide system and application services. We conclude by providing a performance analysis of the protocol versus NFS for sharing a static file system.

Motivation

This paper describes the implementation and use of the Plan 9 design principles and infrastructure under the Linux operating system with the intent of providing a unified ubiquitous distributed environment for the system and applications. Plan 9 [Pike90] was a new research operating system and associated applications suite developed by the Computing Science Research Center of AT&T Bell Laboratories (now a part of Lucent Technologies), the same group that developed UNIX , C, and C++. Plan 9 was initially released in 1993 to universities, and then made generally available in 1995 [9FAQ]. Its core operating systems code laid the foundation for the Inferno Operating System released as a product by Lucent Bell-Labs in 1997 [INF1]. The Inferno venture was the only commercial embodiment of Plan 9 and is currently maintained as a product by Vita Nuova [VITA]. After updated releases in 2000 and 2002, Plan 9 was open-sourced under the OSI [OSI] approved Lucent Public License [LPL] in 2003.

The Plan 9 project was started by Ken Thompson and Rob Pike in 1985. Their intent was to explore potential solutions to some of the shortcomings of UNIX in the face of the widespread use of high-speed networks to connect machines [Pike04]. In UNIX , networking was an afterthought and UNIX clusters became little more than a network of stand-alone systems. Plan 9 was designed from first principles as a seamless distributed system with integrated secure network resource sharing. Applications and services were architected in such a way as to allow for implicit distribution across a cluster of systems. Configuring an environment to use remote application components or services in place of their local equivalent could be achieved with a few simple command line instructions. For the most part, application implementations operated independent of the location of their actual resources.

Commercial operating systems haven't changed much in the 20 years since Plan 9 was conceived. Network and distributed systems support is provided by a patchwork of middle-ware, with an endless number of packages supplying pieces of the puzzle. Matters are complicated by the use of different complicated protocols for individual services, and separate implementations for kernel and application resources. We are proposing leveraging Plan 9's principles in modern commercial operating systems in an attempt to provide a more coherent, unified approach to distributed systems. The rest of this paper reviews Plan 9's principles, discusses the details of the 9P protocol, describes our Linux implementation, and evaluates its performance compared to NFSv3.

Plan 9 Design Principles and Practice

Plan 9's transparent distributed computing environment was the result of three core design principles which permeated all levels of the operating system and application infrastructure:

1) develop a single set of simple, well-defined interfaces to services
2) use a simple protocol to securely distribute the interfaces across any network
3) provide a dynamic hierarchical structure to organize these interfaces

In Plan 9, all system resources and interfaces are represented as files. UNIX pioneered the concept of treating devices as files, providing a simple, clear interface to system hardware. In the 8th edition, this methodology was taken further through the introduction of the /proc synthetic file system to manage user processes [Kill84]. Synthetic file systems are comprised of elements with no physical storage, that is to say the files represented are not present as files on any disk. Instead, operations on the file communicate directly with the sub-system or application providing the service. Linux now contains multiple synthetic file systems representing devices (devfs), process control (procfs), and interfaces to system services and data structures (sysfs).

Plan 9 took the file system metaphor further, using file operations as the simple, well-defined interface to all system and application services. The intuition behind the design was based on the assumption that any programmer knows how to interact with files. As such, interfaces to all kernel subsystems from the networking stack to the graphics frame buffer are represented within synthetic file systems. This unified approach dramatically simplifies access to all system services, evident in Plan 9’s use of only 40 system calls compared to Linux’s 300.

User-space applications and services export their own synthetic file systems in much the same way as the kernel interfaces. Common services such as domain name service (DNS), authentication databases, and window management are all provided as file systems. End-user applications such as editors and e-mail systems export file system interfaces as a means for data exchange and control. The benefits and details of this approach are covered in great detail in the existing Plan 9 papers [PPTTW93].

9P [9man] represents the abstract interface used to access resources under Plan 9. It is somewhat analogous to the VFS layer in Linux [Love03]. In Plan 9, the same protocol operations are used to access both local and remote resources, making the transition from local resources to cluster resources to grid resources completely transparent from an implementation standpoint. Authentication is built into the protocol, and was extended in its Inferno derivative Styx [STYX] to include various forms of encryption and digesting.

It is important to understand that all 9P operations can be associated with different active semantics in synthetic file systems. Traversal of a directory hierarchy may allocate resources, or set locks. Reading or writing data to a file interface may initiate actions on the server, such as when a file acts as a control interface. The dynamic nature of these semantics makes caching dangerous and in-order synchronous execution of file system operations a must.

The 9P protocol itself requires only a reliable, in-order transport mechanism to function. It is commonly used on top of TCP/IP [RFC793], but has also been used over RUDP [RFC1151], PPP [RFC1331], and over raw reliable mechanisms such as the PCI bus, serial port connections, and shared memory. The IL [PrWi95] protocol was designed specifically to provide 9P with a reliable, in order transport on top of an IP stack without the overhead of TCP.

In the fourth edition of Plan 9 released in 2002, 9P was redesigned to address a number of shortcomings and retitled 9P2000 [P903]. 9P2000 removed the file name length limitation of 28 bytes, and sought to optimize several operations involved in traversing file hierarchies. It also introduced a negotiation phase accommodating different versions of the protocol as well as protocol parameter negotiation.

The final key design point is the organization of all local and remote resources into a dynamic private name space. A name space is a mapping of system and application resources to names within a file system hierarchy. Manipulation of the location of the elements within a name space can be used to configure which services to use, to interpose stackable layers onto service interfaces, and to create restricted "sandbox" environments.

Under Plan 9 and Inferno, the name space of each process is unique and dynamic. A name space can be manipulated through mount(1) and bind(1) commands. Mount operations allow a client to add new interfaces and resources to their name space. These resources can be provided by the operating system, by a synthetic file server, or from a remote server. Bind commands allow reorganization of the existing name space, allowing certain services to be "bound" to well-known locations. Bind operations can also be used to substitute one resource for another, for example by binding a remote device over a local one. Binding can also be used to create stackable layers, by interposing one interface over another. Such interposable interfaces are particularly useful for debugging and statistics gathering as in the Plan 9 application iostat(4).

Processes inherit an initial name space from their parent, but changes made to the client’s name space are not typically reflected in the parent’s. This allows each process to have a context-specific name space. The system makes extensive use of this facility to provide context-sensitive file interfaces. For example, in Plan 9, the file /dev/cons refers to the current process' standard input and output file descriptors. In a similar fashion /dev/user reports the user name, and /dev/pid reports the current process id. This same facility is used by the windowing system to provide information and control over the process' window.

Handcrafted name spaces can be used to create secure sandboxes which give users access to very specific system resources. This can be used in much the same way as the UNIX chroot facility - except that the chroot name spaces under Plan 9 can be completely synthetic - with specific executables and interfaces "bound" into place instead of copied to a sub-hierarchy.

The recent open-sourcing of Plan 9 and many of its applications has created the opportunity to leverage some of its unique concepts in other open source operating systems. Alex Viro incorporated the private name space concept into the 2.5 Linux kernel [Love03] and Russ Cox has ported a number of Plan 9 applications and libraries (including libraries allowing the creation of synthetic file server applications) to Linux, BSD, and OSX [plan9ports]. This paper describes the final missing piece, a native 9P protocol for Linux.

The 9P2000 Protocol

As mentioned earlier, 9P2000 is the most recent version of 9P, the Plan 9 distributed resource protocol. It is a typical client/server protocol with request/response semantics for each operation (or transaction). 9P can be used over any reliable, in-order transport. While the most common usage is over pipes on the same machine or over TCP/IP to remote machines, it has been used on a variety of different mediums and encapsulated in several different protocols. The Internet Link (IL) protocol [PrWi95] was a lightweight encapsulation designed specifically for 9P.

9P has 12 basic operations, all of which are initiated by the clients. Each request (or T-message) is satisfied by a single associated response (or R-message). In the case of an error, a special response (R-error) is returned to the client containing a variable length string error message. It is important to note that there are no special operations for directories or links as in VFS [Love03] because these elements are just treated as ordinary files. The operations summarized in the following table fall into three categories: session management, file operations, and meta-data operations.

class	*op-code*	*description*
session	version	parameter negotiation
	auth	security authentication
	attach	establish a connection
	flush	abort a request
	error	return an error
file	walk	lookup pathname
	open	access a file
	create	create & access a file
	read	transfer data from a file
	write	transfer data to a file
	clunk	release a file
metadata	stat	read file attributes
	wstat	modify file attributes

Table 1: 9P2000 Operations

9P is best understood by seeing what messages are transmitted for some standard file system operations. One can do this by using the UNIX 9P server, u9fs, and turning debug mode on. The debug output (which goes to /tmp/u9fs.log by default) displays a human-readable transaction log. What follows in italics is an example session of a Plan 9 client contacting a Unix 9P2000 server and writing to a new file (/tmp/usr/testfile). Explanations of the various operations follow each request/response pair. Messages marked with (→) are from the client to the server, and (←) represent the responses. Note that you will see slightly different transaction sequences from a UNIX client due to the nature of the mapping of VFS operations.

→ *Tversion tag -1 msize 8216 version '9P2000'*

← *Rversion tag -1 msize 8216 version '9P2000'*

The *version* operation initiates the protocol session. The *tag* accompanies all protocol messages and is used to multiplex operations on a single connection. The client selects a unique *tag* for each outbound operation. The *tag* for *version* operations, however, is always set to -1. The next field, *msize* negotiates the maximum packet size with the server including any headers - the server may respond with any number less than or equal to the requested size. The *version* field is a variable length string representing the requested version of the protocol to use. The server may respond with an earlier version, or with an error if there is no earlier version that it can support.

→ *Tauth tag 5 afid 291 uname 'bootes' aname ''*

← *Rerror tag 5 ename 'u9fs authnone: no authentication required'*

The *auth* operation is used to negotiate authentication information. The *afid* represents a special authentication handle, the *uname* (bootes) is the user name attempting the connection and the *aname*, (which in this case is blank), is the mount point the user is trying to authenticate against. A blank *aname* specifies that the root of the file server's hierarchy is to be mounted. In this case, the Plan 9 client is attempting to connect to a Unix server which does not require authentication, so instead of returning an *Rauth* operation validating the authentication, the server returns *Rerror*, and in a variable length strength in the field *ename*, the server returns the reason for the error.

→ *Tattach tag 5 fid 291 afid -1 uname 'bootes' aname ''*

← *Rattach tag 5 qid (0902 1097266316 d)*

The *attach* operation is used to establish a connection with the file server. A *fid* unique identifier is selected by the client to be used as a file handle. A *Fid* is used as the point of reference for almost all 9P operations. They operate much like a UNIX file descriptor, except that they can reference a position in a file hierarchy as well as referencing open files. In this case, the *fid* returned references the root of the server's hierarchy. The *afid* is an authentication handle; in this case it is set to -1 because no authentication was used. *Uname* and *aname* serve the same purposes as described before in the *auth* operation.

The response to the *attach* includes a *qid*, which is a tuple representing the server's unique identifier for the file. The first number in the tuple represents the *qid.path*, which can be thought of as an inode number representing the file. Each file or directory in a file server's hierarchy has exactly one *qid.path*. The second number represents the *qid.version*, which is used to provide a revision for the file in question. Synthetic files by convention have a *qid.version* of 0. *Qid.version* numbers from UNIX file servers are typically a hash of the file's modification time. The final field, *qid.type*, encodes the type of the file. Valid types include directories, append only files (logs), exclusive files (only one client can open at a time), mount points (pipes), authentication files, and normal files.

→ *Twalk tag 5 fid 291 newfid 308 nwname 0*

→ *Rwalk tag 5 nwqid 0*

Walk operations serve two purposes: directory traversal and *fid* cloning. This *walk* demonstrates the latter. Before any operation can proceed, the root file handle (or *fid*) must be cloned. A clone operation can be thought of as a dup, in that it makes a copy of an existing file handle - both of which initially point to the same location in the file hierarchy. The cloned file handle can then be used to traverse the file tree or to perform various operations. In this case the root *fid* (291) is cloned to a new *fid* (308). Note that the client always selects the *fid* numbers. The last field in the request transaction, *nwname*, is used for traversal operations. In this case, no traversal was requested, so it is set to 0. The *nwqid* field in the response is for traversals and is discussed in the next segment.

→ *Twalk tag 5 fid 308 newfid 296 nwname 2 0:tmp 1:usr*

← *Rwalk tag 5 nwqid 2 0:(0034901 1093689656 d) 1: (0074cdd0 1096825323 d)*

Here we see a traversal request walk operation. All traversals also contain a clone operation. The *fid* and *newfid* fields serve the same purpose as described previously. *Nwname* specifies the number of path segments which are attempting to be traversed (in this case 2). The rest of the operands are numbered variable length strings representing the path segments - in this case, traversing to /tmp/usr. The *nwqid* in the response returns the qids for each segment traversed, and should have a qid for each requested path segment in the request. Note that in this case there are two pathname components: the path name is walked at the server, not the client, which is a real performance improvement over systems such as NFS which walk pathnames one component at a time.

→ *Tcreate tag 5 fid 296 perm --rw-rw-rw- mode 1 name 'testfile'*

← *Rcreate tag 5 qid (074cdd4 1097874034) iounit 0*

The *create* operation both creates a new file and opens it. The *open* operation has similar arguments, but doesn't include the *name* or *perm* fields. The *name* field is a variable length string representing the file name to be created. The *perm* field specifies the user, group, and other permissions on the file (read, write, and execute). These are similar to the core permissions on a unix system. The *mode* bit represents the mode with which you want to open the resulting file (read, write, and/or execute). The response contains the *qid* of the newly created (or opened) file and the *iounit*, which specifies the maximum number of bytes which may be read or written before the transaction is split into multiple 9P messages. In this case, a response of 0 indicates that the file's maximum message size matches the session's maximum message size (as specified in the *version* operation).

→ *Tclunk tag 5 fid 308*

← *Rclunk tag 5*

The *clunk* operation is sent to release a file handle. In this case it is releasing the cloned handle to the root of the tree. You'll often see transient *fid*s used for traversals and then discarded. This is even more extreme in the UNIX clients as they only traverse a single path segment at a time, generating a new *fid* for each path segment. These transient *fid*s are a likely candidate for optimization, and may be vestigial from the older 9P specification which had a separate clone operation and didn't allow multiple segment walks.

→ *Twrite tag 5 fid 296 offset 0 count 8 'test'*

← *Rwrite tag 5 count 8*

We finally come to an actual I/O operation, a write operation that writes the string 'test' into the new file. *Write* and *read* operands are very similar and straightforward. The *offset* field specifies the offset into the file to perform the operation. There is no separate seek operation in 9P. The *count* represents the number of bytes to read or write, and the variable length string ('test') is the value to be written. The response *count* reports the number of bytes successfully written. In a *read* operation the response would also contain a variable length string of *count* size with the data read.

→ *Tclunk tag 5 fid 296*

← *Rclunk tag 5*

This final clunk releases the *fid* handle to the file -- approximating a close operation. You'll note that the only *fid* remaining open is the root *fid* which remains until the file system is unmounted. Several operations were not covered in this transaction summary. *Flush* is almost never used by clients in normal operation, and is typically used to recover from error cases. The *stat* operation, similar to its UNIX counterpart, is used to retrieve file metadata. *Twstat* is used to set file metadata, and is also used to rename files (file names are considered part of the metadata).

9P2000 Unix Extensions

Many modern UNIX systems, including Linux use a virtual file system (VFS) layer as a basic level of abstraction for accessing underlying implementations. Implementing 9P2000 under Linux is a matter of mapping VFS operations to their associated 9P operations. The problem, however, is that 9P2000 was designed for a non-UNIX system so there are several fundamental differences in the functional semantics provided by 9P.

Under Plan 9, user names as well as groups are represented by strings, while on Unix they are represented by unique numbers. This is complicated by Linux making it exceedingly difficult to map these numeric identifiers to their string values in the kernel. Many of the available UNIX network file systems avoid this issue and simply use numeric identifiers over the wire, hoping they map to the remote system. NFSv4 [NFS4] has provisions for sending string group and user info over the wire and then contacting a user-space daemon which attempts to provide a valid mapping.

We use two different name mapping approaches based on server system type. When contacting a 9P server on another UNIX system we use numeric identifiers. When contacting a Plan 9 file server we map all numeric ids to the numeric id of the mounting user. While this yields less than accurate uid/gid information in directory listings, permissions checking (which is done on the remote server) is still valid. A potential future piece of work would be to provide a user-space daemon similar to NFSv4 to provide uid/gid mapping services or perhaps somehow leverage the existing NFSv4 service.

One of the unique aspects of the Plan 9 name space is that it is dynamic. Users are able to bind files and directories over each other in stackable layers similar to union file systems. This aspect of Plan 9 name spaces has obviated the need for symbolic or hard links. Symlinks on a remote UNIX file server will be traversed transparently as part of a walk - there is no native ability within Plan 9 to create symlinks. This breaks many assumptions for Linux file-systems and many existing applications (for example the kernel build creates a symlink in the include directory as part of the make process).

To preserve compatibility with these existing applications we implemented a transparent extension to the file system semantics which doesn't effect the protocol syntax. Files requiring this extension have their filenames prefixed with a marker character, '/', which is normally illegal in file system operations. A string following the '/' provides details of the type of extension and is followed by a numeric mode extension. For symlinks and hard links the file's content contains the link information. These extensions are essentially ignored by the Plan 9 file server, but interpreted correctly by UNIX clients and servers.

The same extension can be used to provide support for additional mode bits and other file types not provided under Plan 9. The extension codes are documented in the table below:

extension	***code***
symbolic link	/syml*x*
hard link	/link*x*
character device	/char*x*
block device	/blck*x*
pipe	/pipe*x*
extra mode	/mode*x*
mode	***code(x)***
none	0
set-user-id	1
sticky bit	2
directory exec (X)	4

Table 2: 9P2000.u Extension Codes

Since these are all extensions to the 9P2000 protocol, they must be negotiated during the version stage of establishing the connection. The clients register interest in the extension by appending a *.u* to the version string (e.g. 9P2000.u). If the server is capable of providing the UNIX extensions, it will respond with a 9P2000.u in the response message. If the server does not wish (or cannot) provide the UNIX extensions, it will respond without the .u (e.g. 9P2000).

An operation which does not currently have any support in the 9P2000 protocol or UNIX extensions is ioctl. Fortunately, in the Linux world, ioctl is being deprecated in favor of sysfs-based mechanisms. For interfaces still requiring ioctl operations (such as sockets), gateway synthetic file server applications can map ioctl functionality to synthetic control files.

Details of the Linux Implementation

For the most part, 9P2000 can be directly mapped to the Linux VFS interface. There are several semantic differences beyond the syntactic differences mentioned in the above section which are worth noting.

One difference is the model by which a file system is mounted. On UNIX systems this is done either at boot time, by the super user, or through an automounter (which must be configured for particular resources by the super user). Additionally, once a file system is mounted it is visible (and accessible) to all users. By contrast, under Plan 9 file systems are mounted by individual users into their private name space environment. The connection to the file server is authenticated by the credentials of the individual user at mount time, and the file system is mounted in the user's private name space - so it is not directly visible to any other user. Even diskless terminals work this way under Plan 9 - a user must first log in before the terminal mounts its root file system from the file server.

The Linux 9P2000 driver can accommodate both systems. It can mount the file server as root (or some other special user) into the global Linux name space or it can operate with the user initiating the mount and it being authenticated against his or her user ID. The two complications are that, by default, Linux has a global name space for all users, and ordinary users don't typically have permission to mount arbitrary systems on arbitrary mount points.

This can be solved by having a special set-uid version of mount which allows certain users (or users in a particular group) to mount from certain systems matching a list of regular expressions to particular mount points matching a regular expressions. A special version of mount is required in order to do DNS resolution of the server names as NFS support for DNS resolution is a special case implementation in the standard mount application. Users can spawn off a private name space by using a simple wrapper utility (as described in the *Application* section).

A third alternative, which is currently the most secure model, is for the user to ssh to the remote file server and start his own private u9fs instance which uses the ssh connection as a transport. On the client system a set-uid mount application can attach to the i/o stream and a wrapper can be used to establish a private name space if desired. This still requires a properly configured set-uid mount application on the client and is only useful after the client system is fully booted. In fact, this is the way that Plan 9 users commonly mount UNIX file systems using u9fs when they don't have root permissions on the file server.

File creation under Plan 9 is atomic. That is to say the file is created and opened in a single 9P operation. Under VFS, creation semantics are entirely separate from open, and therefore not atomic. The Linux 9P2000 implementation seeks to preserve some of the atomicity of the original Plan 9 semantics by caching the fid in the inode structure - however, this does not guarantee the same atomic semantic.

Cache policy was a major concern when implementing the 9P2000 driver. Linux file systems routinely use two caches: the dcache provides caching of directory lookup information effectively caching inodes and metadata so they don't have to be re-read from the disk, while the page cache provides a data cache. Both create problems as 9P2000 is intended to be a non-cached synchronous protocol since it does more than just provide a transport to share files.

The dcache creates problems because it caches inodes, effectively caching both metadata and file/directory lookup. The problem is that synthetic file system semantics may be attached to metadata operations (stat/twstat) and file system hierarchy movement (walk). This case is particularly important to consider due to the dynamic nature of synthetic file systems. For example, since a file server assumes it will see walks every time a directory is traversed, it could use such traversal as a locking mechanism. If the results of the walk is cached and reused, it would violate the lock semantics. The other problem with the dcache is that *fids* stay open (attached to the inode in the dcache) which would otherwise be clunked. Clunks are another operation which commonly have associated semantics within synthetic file systems (such as unlocking).

One solution is to always report dcache entries as invalid, forcing the clunk of their associated *fids* and requiring a rewalk from the mount-point to their location. This is unfortunate as the directory cache provides a nice performance improvement, particularly in cases where a deep directory structure is being repeatedly traversed. An alternative solution lies in the version field of Plan 9 *qid*s. By convention, static file versions start at 1 and synthetic files always have version 0. So the dcache solution is to automatically invalidate any dcache entry with a *qid.version* equal to 0, otherwise validate the dentry in a normal fashion via a stat transaction and comparing the version numbers.

The same basic policy can be applied to caching file system data in the page cache. Since synthetic file systems typically do not have version or even modification date information, there is no way to validate the cached inode, so you must assume it is invalid. Typically it is not desirable to cache synthetic file systems anyways, it doesn't make sense to think about caching the dynamic data in the /proc file system.

Loose consistency models present another particularly difficult problem. For example, by default NFS (v2 & v3) employs a timeout based invalidation strategy for its page cache and also implements a time-delayed write back in an attempt to coalesce operations. This results in admirable write performance optimization but undermines any synchronous operation of the protocol. This could be particularly damaging when you are using writes to the file system to control a remote service or manage locks.

Because of the drastically different semantics, dcache, page cache, and delayed write-back caching have been removed from the Linux 9P2000 driver. Plan 9 itself uses a stackable caching file system to provide a similar level of cache optimization. Unlike the standard Linux page cache, the Plan 9 cache file system can even use a disk as a backing store. This is particularly effective for workstations that have a disk but get their root file systems from a server. David Howell's CacheFS [CFS] shows promise as providing a similar stackable service for Linux. Ultimately, it is our intent to provide some level of cache for 9P via integration with a stackable cache system while preserving correct synthetic file system semantics.

Applications

9P2000 support provides a nice alternative to NFS for distributing static files. It can be configured in such a way that users can export and mount their own file system resources. Its synchronous mode of operation and optional caching make it ideal for situations where the cache models and delayed writes of other file systems cause problems. Many utilities, such as CVS and various e-mail servers, discourage the use of NFS repositories due to concerns with data corruption resulting from bad cache behavior and lack of transaction semantics.

However, as mentioned several times before - distributed file service is not the only benefit of 9P2000. It can be used to share networking stacks and block and character devices between members of a 9P2000 cluster. It can be used to manipulate Linux synthetic file systems, such as /proc and /sys providing distributed control and management. It also can be used as an infrastructure to implement distributed applications.

A key to effective use of 9P2000 is the recently added private name space extensions in the 2.4.19 and later kernels. In order to use these, a special flag (*CLONE_NEWNS*) needs to set when the *clone* system call is used to create a child process. When the flag is used, Linux will create a copy of the parents name space instead of sharing the same copy. Modifications to the child's name space will not be visible to the parent and vice-versa. This can be used by normal users by writing a simple wrapper application for the standard shell which creates a new name space each time a new shell is invoked. An example wrapper application is included in the v9fs distribution.

Another possible use of the 9P2000 Linux support is with Russ Cox's UNIX ports of the Plan 9 applications. Several of the applications, including ACME [Pike94] and plumbing [Pike00], export synthetic file system interfaces to application structures. The plan9ports package includes wrapper applications which can be used with synthetic file systems without mounting them, but, having a native 9P file system, make access and manipulation much more easy and intuitive.

Similar types of synthetic file systems can now be written for Linux applications. An example would be a port of the task bag file system. The task bag file system is a synthetic file system that presents users with three directories: questions, working, and answers. Programs needing work done create exclusive-open files in the questions directory and write the questions to them, one question to each file. Once the file is closed, workers can pick up work from the questions directory (by opening a file in that directory). Files that are opened a second time for writing (i.e. by a worker) will disappear from the questions directory and appear in the working directory. If the file is closed without a write, it means the worker died or gave up; it reappears in the questions directory. When the file is closed, if there was data written to it, the file name will now appear in the answers directory.

This file system models an earlier system built using SunRPC. In that system, programs that wished to use the taskbag need to be built with SunRPC. In the taskbag file system, it is possible to use shell scripts to create, acquire, and get the answers for tasks. Status can be determined with 'ls' and 'cat'. The system works transparently over a Grid [9grid]. All in all, the task bag file system is far more capable than the SunRPC taskbag system that it replaces.

Private name spaces have also been integrated into the Clustermatic software suite [Clus]. Users can specify a set of mount points that are to be activated when their process or its children are run on a cluster node. When the process is migrated to the cluster node, the mount points are set up by the kernel before the process starts and are removed by the kernel after the process and any children exit. Note that this is not really an automounter, though it behaves in a similar way. The mount points are private and they are part of the context of the process, not defined a file in /etc. We have tested this system on the 1024-node Pink cluster at LANL. 1024 mounts take 20 seconds to set up, but once the system is running it is faster than NFS and far more stable.

Performance

As mentioned earlier, 9P is a unifying protocol - it combines methods for name space organization, resource sharing, distributed applications, and file service. We focus our performance characterization on the distributed file service since it has easily identified comparison points and a host of generally available benchmarks. We chose to compare our implementation to NFS, because it seems to be the most widely used and understood distributed file system protocol.

Our test environment is a cluster of identically configured dual-processor 866 MHz Pentium III servers, each configured with 256 MB of memory, a 36GB IBM DDYS-T36950N SCSI drive formated with an ext2 file system, an Alteon 1 GB Ethernet card, and running Linux 2.6.8. We will show evaluations with caches disabled within 9P2000 and will test against NFSv3 (both tcp and udp). We compared the protocols with 8k and 32k buffer sizes.

We used two evaluation benchmarks to characterize the performance of our 9P2000 driver and compare it to NFS. The first is the Bonnie [Berry90] suite of file system benchmarks which test overall throughput through operations on a single large file. The second is the PostMark [Katch97] benchmark from Network Appliance; it performs a large number of transactions with smaller block operations and many files.

The Bonnie benchmark performs a series of tests on a single file of a specified size. The default is a 100 MB file, but we followed the instruction's suggested guideline of twice the size of available DRAM (512MB). This limits the effectiveness of cache operations and exposes the performance of the underlying file operations. It performs sequential writes and reads of the file, both a character at a time and in blocks. It also performs a read, modify, write sequence (rewrite), and random seeks followed by reads and writes of small chunks of data.

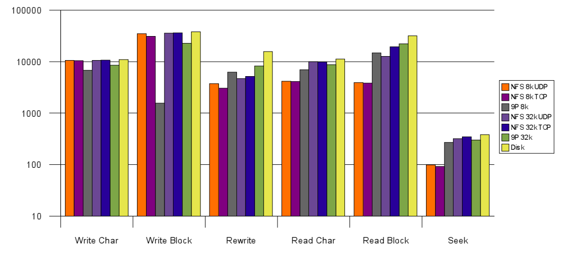


Illustration 1: Bonnie Results (KB/sec)

The results show comparable or better performance for all operations except write operations with small block sizes. On write benchmarks, 9P2000 does poorly due to the fact that async NFS delays write-back to coalesce transactions and does a better job of pipelining write operations. The effect of the operation coalescing is somewhat mitigated with larger block sizes, but the pipelining of write requests still gives NFS a distinct advantage.

For other operations 9P2000 shows slight advantages over NFS due to the lower complexity of the protocol and the fact this it lets the server handle metadata update instead of issuing separate transactions. 9P's lower complexity can be seen in its implementation with only 12 operations and just under 3500 lines of code, while NFS has 17 operations and is implemented in 9357 lines of code (source code counts based on David Wheeler's SLOCCount).

With the larger working set there is significantly less memory for caches, reducing the effectiveness of NFS's loose consistency model. Runs which had smaller footprints showed dramatic advantages for NFS, particularly for read block operations. Caches clearly have advantages and need to be considered for distributing static file systems.

Analysis of protocol traffic during the benchmark (using NFSDump [LISA03] and 9P server logs) shows approximately 50% fewer operations for NFS with 32k buffer sizes. This dramatic drop in the number of operations (primarily read and write operations) shows the effectiveness of caches for operation coalescing (in the write-char and read-char portions of the benchmark). With 8k buffer sizes, NFS retains an advantage in fewer write operations, but has roughly the same number of read operations as 9P. The fact that this huge reduction in the number of operations over the wire doesn't have a more dramatic effect on the performance seems to indicate the overhead added by NFS coherence and the page cache are detrimental to performance of workloads with poor locality.

PostMark is a benchmark developed under contract to Network Appliance. Its intent is to measure the performance of a system used to support a busy email service or the execution of a large number of CGI scripts processing forms. It creates the specified number of files randomly distributing the range of file sizes: it is typically used with very large numbers of relatively small files. It then runs transactions on these files, randomly creating and deleting them or reading and appending to them, and then it deletes the files. We ran our PostMark configuration with 500 files and 50000 transactions. This seemed sufficient to differentiate between the file systems and had a tractable run-time.

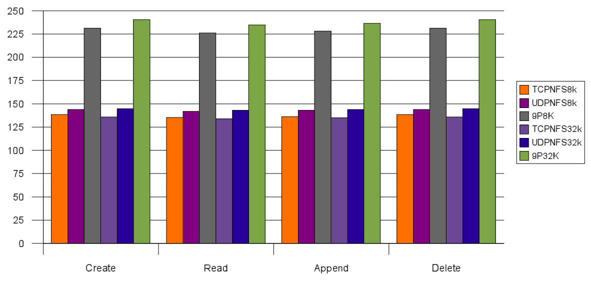


Illustration 2: PostMark Operations (op/sec)

PostMark paints a very different picture than the Bonnie benchmark, since the files and operations are much smaller - low latency handling of requests is much more important than overall throughput. 9P2000 demonstrates approximately double the performance of TCP and UDP NFSv3 (which effectively scored the same for the PostMark benchmark). Analysis of the protocol traffic shows 9P2000 with half the operations of NFS for 8k and 32k buffer sizes. NFS still shows an advantage on the number of read and write operations (in fact never sending a read operation over the wire due to the effectiveness of the page cache). However, NFS sends more than twice as many lookup operations and sends access messages almost every time it touches the files. The fact that so many synchronous operations are required to access a file really inhibits performance for NFS. By contrast, 9P does all permissions checking on the server and requires only a single operation to create and open a file.

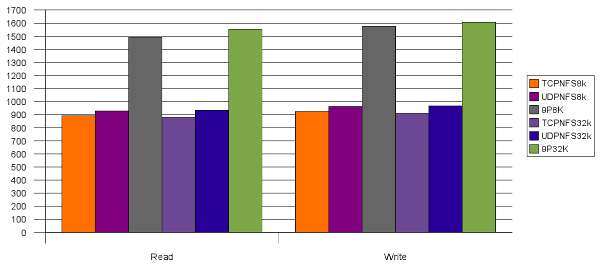


Illustration 3: PostMark Bandwidth (KB/sec)

Related Work

9P is nominally just a remote file system. However, due to the unique way in which it is used to distribute not just files but other system and application resources, it has advantages over other file system protocols for certain applications.

The Linux kernel supports several different network file systems. There is experimental support for the Andrew File System [HOW88], Coda [SATY90], Microsoft SMB and Common Internet File System [CIFS], and Netware's NCP [Majo94]. The most commonly used is the NFS protocol [Sun89]. Linux supports NFSv3, experimental support for NFSv4 [NFS4] and NFS over TCP. All of these are targeted purely for distributing static files and don't implement the correct cache behavior or transaction semantics to support more complicated distribution of devices, system resources, or an easy mechanism for applications to export synthetic file systems. Most of these protocols are tied to either TCP/IP or UDP/IP. While some support other transports, such as RDMA interfaces, none of these variants are in common use today.

The Linux network block device [NBD00], and its cousin the Ethernet block device [EBD02], and iSCSI [RFC3720] provide remote access to block and SCSI drivers. However, they provide no solution for various other system elements and don't support an organizational name space or a mechanism for coherent multi-user access.

As an application interface, 9P has many contemporaries. Remote Procedure Call (RPC) [Birr94] provides the closest contemporary providing both application level servers and a foundation layer for NFS -- however, it doesn't provide any of the semantics necessary to structure a functional name space.

Another interesting area to evaluate 9P is within cluster resource distribution and management. 9P provides a single protocol which can enable sharing of devices, file systems, and compute resources while also providing a convenience name space to organize these resources. While many other infrastructures attempt to provide these facilities [Fos96][Litz88], none provide the transparent ubiquity of Plan 9's model or an unified solution to resource distribution and management in a single protocol.

Future Work

The largest missing piece from the Linux 9P2000 implementation is security. There is some rudimentary security provided by u9fs - either by using a rhosts file to specify what hosts are allowed to connect (similar to a stripped down exports file), or through a simple insecure challenge-response password system. Those concerned with security that want to use the existing implementation must tunnel through an ssh system which gives both user-authentication and data security.

However, Plan 9 has a rich set of security mechanisms which neither the Linux client driver or server use. One piece of future work would be to integrate these security mechanisms, plus the ability to encrypt and digest messages into the Linux drivers. Alternatively, or perhaps additionally, one could integrate Linux-style authentication schemes such as Kerberos and/or LDAP.

Another piece of future work would be to enable and tune 9P2000 to run on other transports beyond TCP/IP and pipes. Particularly interesting would be a port of 9P2000 to an RDMA [Mogu04] interface without an IP encapsulation. There is already an effort underway to port NFS to RDMA [Call02] and implement special RDMA file systems [Talp03].

NFS has the advantage of kernel-mode server. A port of the u9fs server application into the kernel with best efforts towards a zero-copy infrastructure should significantly increase the performance of the 9P2000 Linux implementation.

With the 9P2000 infrastructure now in place for Linux, a whole host of applications and synthetic file system gateways can be developed to provide Plan 9-style services and resource sharing. Gateway applications and drivers need to be developed for the IP stack, graphics systems, standard I/O console and other resources not currently exported to the Linux file system. Proxy drivers could be written to gateway character, block, and network driver operations across 9P2000 to devices on remote servers. Helper applications, like 9fs(1) and cpu(1) in Plan 9, need to be ported to Linux to allow easy access and use of the cluster resources made available with 9P2000. Finally, existing Plan 9 application ports need to be updated to use the native 9P file system support.

Conclusions

Plan 9 was developed for distributed systems with three design principles in mind: represent all system and application resources as files, distribute those files using a simple protocol, and organize these distributed resources in a dynamic per-process private name space. With support for private name spaces and the implementation of 9P2000 now in the Linux 2.6 kernel, we can explore Plan 9 inspired distributed resource sharing and infrastructure within a commercial operating system.

The PostMark benchmark shows that 9P2000 performance for typical real-world workloads is superior to NFS, while the Bonnie benchmark shows NFS gets a significant benefit from time-delay write-back and loose read consistency on large static files. While performance for static files isn't a prime motivation for the 9P2000 protocol, we are confident that a loose consistency cache-layer similar to NFS could be implemented which would yield similar, if not better performance results. The performance results suggest that even without this cache layer, 9P2000 is a superior protocol for performing synchronous distributed file operations and demonstrates better performance for smaller files.

However, performance isn't the main advantage of 9P2000. The important thing to understand is the new paradigm of unified system resource sharing and distributed application design that it enables. The qualitative advantages of such a system have been documented in both the Plan 9 and Inferno technical papers. They can be seen, to a limited extent, in the synthetic file systems currently available under Linux (e.g. /proc and /sys). It is our hope that the availability of the paradigm in a widely available commercial operating system such as Linux will inspire more developers to experiment with this approach to distributed computing. The source code is available from http://v9fs.sourceforge.net under the GPL license.

Acknowledgments

The 2.6 port of V9FS and performance analysis was supported in part by the Defense Advance Research Projects Agency under Contract No. NBCH30390004. The original V9FS research work by Ron Minnich was supported by DARPA Contract #F30602-96-C-0297.

Research conducted in the Cluster Research Lab at the Los Alamos National Labs was funded in part by the Mathematical Information and Computer Sciences (MICS) Program of the DOE Office of Science and the Los Alamos Computer Science Institute (ASCI Institutes). Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. Los Alamos, NM 87545 LANL LA-UR-04-7478.

References

[9FAQ] "Plan 9 from Bell Labs FAQ", http://ask.km.ru/3p/plan9faq.html.

[9grid] A. Mirtchovski, R. Simmonds, R. Minnich, "Plan 9 -- an Integrated Approach to Grid Computing" International Parallel and Distributed Processing Symposium April 2004.

[9man] Plan 9 Programmer's Manual, Volume 1, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[Berry90] M. Berry, "Bonnie Source Code" http://www.textuality.com/bonnie/intro.html ,1990.

[Birr94] A. D. Birrell, B. J. Nelson, "Implementing Remote Procedure Calls", Proceedings of the ACM Symposium on Operating System Principles 1984.

[Call02] B. Callaghan, "NFS over RDMA" Proceedings of FAST 2002.

[CFS] David Howell "CacheFS" http://www.redhat.com/archives/linux-cachefs ,October 2004.

[CIFS] C. Hertel, "Implementing CIFS: The Common Internet File System", Prentice Hall PTR September 2003.

[Clus] "Clustermatic: A complete cluster solution", http://www.clustermatic.org

[EBD02] E. Van Hensbergen, F. Rawson, "Revisiting Link-Layer Storage Networking", IBM Technical Report #RC22602 2002.

[Fos96] I. Foster, C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit" The International Journal of Supercomputer Applications and High Performance Computing, 1996.

[HOW88] J.H. Howard, "An Overview of the Andrew FIle System", Proceedings of the USENIX Winter Technical Conference, Feb 1988.

[INF1] S.M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey, and P. Winterbottom, "The Inferno Operating System", Bell Labs Technical Journal Vol. 2, No. 1, Winter 1997.

[Katch97] J. Katcher, "PostMark: a New Filesystem Benchmark." Technical Report TR3022, Network Appliance 1997.

[Kill84] Tom Killian, "Processes as Files", USENIX Summer Conf. Proc. , Salt Lake City June, 1984.

[LISA03] D. Ellard and M. Seltzer, "New NFS Tracing Tools and Techniques for System Analysis" Large Installation System Administration Conference, 2003.

[Litz88] M. Litzkow, M. Livny, M. Mutka, "Condor: A Hunter of Idle Workstations" Proceedings of the 8th

International Conference of Distributed Computing Systems", 1988.

[Love03] Robert Love, “Linux Kernel Development”, Sams Publishing, 800 E. 96th Street, Indianapolis, Indiana 46240 August 2003.

[LPL] “Lucent Public License Version 1.02”, http://cm.bell-labs.com/plan9dist/license.html

[Majo94] D. Major, G. Minshall, and K. Powell, “An Overview of the NetWare Operating System”, Proceedings of the 1994 WInter USENIX Pages 355-72, January 1994.

[Mogu04] J. Mogul, “TCP offload is a dumb idea whos time has come” Proceedings of HotOS IX 2004.

[NBD00] P. T. Breuer, A. Marin Lopez, and A. G. Ares, “The Network Block Device”, Linux Journal Issue 73 May 2000.

[NFS4] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, et. al, “The NFS Version 4 Protocol” Proceedings on the 2nd international system administration and networking conference, 2000.

[OSI] “Open Source Initiative”, http://www.opensource.org"

[P903] “Plan 9 From Bell Labs Fourth Release Notes”, http://plan9.bell-labs.com/sys/doc/release4.html , June 2003.

[Pike00] Rob Pike, “Plumbing and Other Utilities”, Plan 9 Progreammer's Manual Vol 2. pp 219-234.

[Pike04] R. Pike, “Rob Pike Reponds” Slashdot Interview October 18, 2004.

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, “Plan 9 from Bell Labs”, UKUUG Proc. of the Summer 1990 Conf., London, England, 1990.

[Pike91] Rob Pike, “8½, the Plan 9 Window System”, USENIX Summer Conf. Proc., Nashville, June, 1991, pp. 257-265

[Pike94] Rob Pike, “Acme: A User Interface for Programmers”, USENIX Proc. of the Winter 1994 Conf., San Francisco, CA.

[plan9port] Russ Cox, “Plan 9 from User Space”, http://swtch.com/plan9port

[PPTTW93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, “The Use of Name Spaces in Plan 9”, Op. Sys. Rev., Vol. 27, No. 2, April 1993, pp. 72-76.

[PrWi95] Dave Presotto and Phil Winterbottom, “The IL Protocol”, Plan 9 Programmer's Manual, Volume 2, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[RFC793] RFC793, Transmission Control Protocol, DARPA Internet Program Protocol Specification, September 1981.

[RFC1151] RFC1151, Reliable User Datagram Protocol (version 2), Internet Engineering Task Force, April 1990.

[RFC1331] RFC1331, The Point-to-Point Protocol, Internet Engineering Task Force, May 1992.

[RFC3720] RFC3720, Internet Small Computer Systems Interface, Internet Engineering Task Force, April 2004.

[SATY90] M. Satyanarayanan, “Coda: A Highly Available File System for a Distributed Workstation Environment” IEEE Transactions on Computers, 1990.

[Sun89] Sun Microsystems, “NFS: Network file system protocol specification”, RFC 1094, Network Information Center, SRI International, March, 1989.

[Talp03] T. Talpey, “The Direct Access File System” Proceedings of FAST 2003.

[thomo95] Ken Thompson, “The Plan 9 File Server”, Plan 9 Programmer's Manual, Volume 2, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[VITA] “Vita Nuova Home Page”, http://www.vitanuova.com

[v9fs] Ron Minnich, “Plan 9-style File System for Linux/BSD”, http://sourceforge.net/projects/v9fs

Ourmon and Network Monitoring Performance

James R. Binkley   Bart Massey
Computer Science Dept.
Portland State University
Portland, OR, USA
{*jrb,bart*}*@cs.pdx.edu*

Abstract

Ourmon is an open-source network management and anomaly detection system that has been developed over a period of several years at Portland State University. Ourmon monitors a target network both to highlight abnormal network traf c and measure normal traf c loads. In this paper, we describe the features and performance characteristics of Ourmon.

Ourmon features include a novel mechanism for running multiple concurrent Berkeley Packet Filter (BPF) expressions bound to a single RRDTOOL-style graph, as well as various types of top talker (top-N) lters aimed at conventional network ow measurements and anomaly detection. These features permit a variety of useful and easily-understood measurements.

One problem that sniffer-based network monitor systems face is network-intensive attacks that can overwhelm monitoring and analysis resources. Lab experiments with an IXIA high-speed packet generator, as well as experiences with Ourmon in a real network environment, demonstrate this problem. Some recent modi cations to Ourmon have greatly improved its performance. However, minimum-size packets in a high-speed network can still easily make a host lose packets even at relatively slow rates and low monitor workloads. We contend that small packet performance is a general network security problem faced by current monitoring systems including both open source systems such as Ourmon and Snort, and commercial systems.

1 Introduction

The Ourmon [15] network monitoring system is an open-source tool for real-time monitoring and measurement of traf c characteristics of a computer network. It runs on FreeBSD, and Linux. (There is also code for Solaris, although it is currently unmaintained.) Ourmon's feature set includes various top talker lters and multiple instances of the Berkeley Packet Filter (BPF) which taken together allow us to capture interesting features of the envelope of incoming IP packets. Network monitoring and data visualization are typically performed on separate hosts. The data visualization system uses standard network graphical tools to display the resulting measurements in a fashion that highlights anomalies.

The Internet has recently faced an increasing number of bandwidth-intensive Denial-Of-Service (DOS) attacks. For example, in January 2003 the Slammer worm [4, 12] caused serious disruption. Slammer not only wasted bandwidth and affected reachability, but also seriously impacted the core routing infrastructure. At Portland State University (PSU), four lab servers with 100 Mb NIC cards were infected simultaneously. These servers then sent approximately 360 Mb/s of small packets to random destinations outside of PSU. This attack clogged PSU's external connection to the Internet, in the process also causing important network monitoring failures. Due to the semi-random nature of the IP destination addresses generated by the worm, the CPU utilization of a router sitting between network engineers and network instrumentation rose to 100%. Engineers were thus cut off from central network instrumentation at the start of the attack.

We recently acquired an IXIA 1600 high-speed packet generator. The Slammer attack inspired us to test our Ourmon network monitoring system against a set of Gigabit Ethernet (GigE) ows. Our test ows included maximum-sized (1518 byte) and minimum-sized (64 byte) UDP packets, with both xed and rolling IP destination addresses.

The Ourmon network measurement system architecture consists of two parts: a front-end *probe* and a back-end *graphics engine* system. Optimally these two parts should run on two separate computers in order to minimize the application compute load on the probe itself. Our goal in these experiments has been to test the performance of our probe and its BPF network tap rather than the back-end system.

We constructed a test system consisting of: the IXIA with two GigE ports; a line speed GigE switch capable of port-mirroring; and a FreeBSD workstation with a GigE NIC card. The IXIA was set up to send packets from one GigE port to the other. The switch was set up to mirror packets from one IXIA port to the UNIX host running our front-end probe.

Like other tools including tcpdump [19], Snort[16], or Ntop [5, 14], the Ourmon front-end uses the BPF as a *packet tap*. The application takes a stream of unfiltered packets directly from a BPF kernel buffer fed by an Ethernet device, bypassing the host TCP/IP stack. The interface interrupts on packet input, and hands the trimmed packet (containing all headers through layer 4) to the kernel BPF filter buffer. The Ourmon probe application reads packets, subjecting each packet in turn to a set of configuration filters. It thus makes sense to separately test the performance of the BPF and the performance of the Ourmon probe filter system.

Our experimental questions include the following:

1. Using GigE with maximum or minimum-sized packets, at what bit rate can the underlying packet tap and buffer system successfully process all packets?
2. Using GigE with maximum or minimum-sized packets, what is the smallest BPF kernel buffer size (if any) for which all packets are successfully processed?
3. Ourmon has three kinds of filters: hardwired C filters, BPF-based interpreted filters, and a "top-N" flow analysis system (one of a set of top-N tuple types). Can we determine anything about the relative performance of these filters? If we are receiving a high number of packets per second, which of these filters can keep up?
4. With the Slammer worm, we know that semi-random IP destinations led to inefficient route caching in intermediary routers. What happens when we subject our top-N flow filter to rolling or semi-random IP destinations?

In section 2 we provide a short introduction to the Ourmon system. In section 3 we discuss our test setup. In section 4 we present test results. In section 5 we discuss possible means for improving our performance, including several small-scale application optimizations that have shown reasonable performance improvements. In section 6 we present related work. Section 7 contains a discussion of problems and future work. Section 8 draws some brief conclusions.

2 Introduction to Ourmon

Our measurement work in subsequent sections focuses on the Ourmon probe rather than the graphics engine. However, in this section we give an overview of the complete Ourmon system. This serves to introduce Ourmon and to describe the basis of the measurement effort. To further these goals, we discuss the system at a high-level, introducing the basic feature sets which we call *filters*.

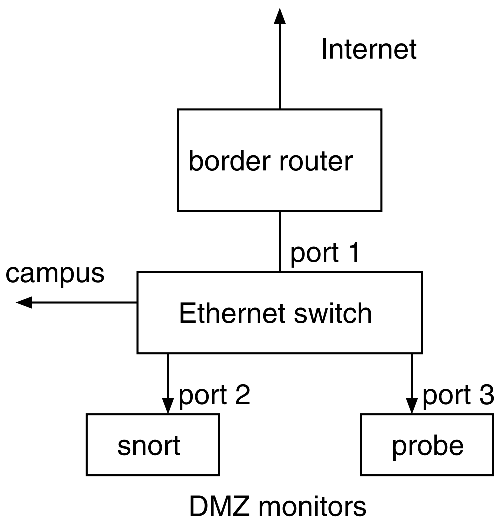


Figure 1: Ourmon network setup

We provide some probe configuration examples and a few sample visualizations. All visualizations have been taken from actual data produced from the Ourmon probe stationed in PSU's "network center" or DMZ. Ourmon is a complex system. While detailed workings of every feature of the Ourmon system are outside the scope of this paper, we attempt to give an overall understanding of system operation.

2.1 Architecture

Ourmon is a "near" real-time web-based network monitor. Web-based data never lags reality by greater than one minute. The system (and its name) are inspired by SNMP RMON [22] monitors. The Ourmon probe assumes the port-mirroring functionality of Ethernet-based switches. A typical setup may be seen in Figure 1. Ourmon can be configured in many ways. At PSU, the Ourmon probe is placed in an internet gateway network so that we can see all traffic going to and from the Internet. In addition, within the PSU Maseeh College of Engineering and Computer Science, we use an Ethernet switch and set Ourmon up to watch important server traffic. An Ethernet switch is configured to mirror (duplicate) packets sent to its Internet connection on port 1. All packets received via the Internet port are copied to port 3, which is running the front-end Ourmon probe on a FreeBSD system using the BPF packet tap. Thus the probe setup is similar to that of Snort, which we show running on port 2 of the switch. The back-end graphics engine is not performance critical. It may run on a second computer, which need not be exposed to the Internet.

The probe, written in C, has an input configuration file and a main output statistics file. (Depending on the features used, other output files are possible.) The configuration file, *ourmon.conf*, specifies various named filters for the probe to use. Typically probe output is written to a small ASCII file, *mon.lite*, that summarizes the last 30 seconds of filter activity in terms of statistics recorded by each configured filter. Ourmon takes copious packet data and tries to summarize it in a statistical way, typically producing integers bound to BPF-based filters or lists of tuples. Tuples may be top-N flows or other top-N tuples typically keyed to an IP source "host". The goal is to produce small amounts of heavily summarized output data. All probe inputs and outputs are in simply-formatted ASCII, facilitating analysis and tool-based processing. If the graphics-engine is on a separate computer, the resulting output files may be copied over the network to that box. The graphics engine, in turn, produces various graphic outputs and ASCII reports for web display. This file transfer is a simple task, accomplished using standard UNIX tools. One typically uses a batch ssh script driven by crontab to accomplish the transfer by pulling data from the probe. Other file transfer programs including rsync, wget, or even NFS will also work. The probe is protected from unauthorized access via a host-based access control list.

The graphics engine, written in Perl, produces several kinds of graphics and reports. RRDTOOL-based [17] strip charts are used with BPF *filter-sets* and hardwired filters. A filter-set is a set of BPF expressions bound to a single RRDTOOL graph. RRDTOOL graphs are wrapped in HTML web pages to ease access. Web pages constituting a year of baselined data are available via the RRD logging system. Histograms and reports are used to display the top-N flow filter and other similar ”top talker” tuple lists. A variety of logging is performed: raw logging of probe output; individual logs per top-N tuple; and report summarizers for some of the more interesting tuples. The resulting reports provide both hourly summaries for the current day, and summary data for the last week. Ourmon does not include a web server (we typically use apache). Our basic install provides a default installation of BPF-based filter-sets and top talker web pages, with individual web pages per filter, as well as a top-level web page for accessing the data.

Figure 2 shows the overall Ourmon system architecture. Upon installation, our configuration utility creates probe and graphics engine shell scripts. In the back-end graphic box, it also installs a set of static web pages which encapsulate runtime generated graphics and reports. The probe is started and stopped via its shell script and runs as a background daemon. It typically runs at boot. It takes the *ourmon.conf* input file, parses it, and

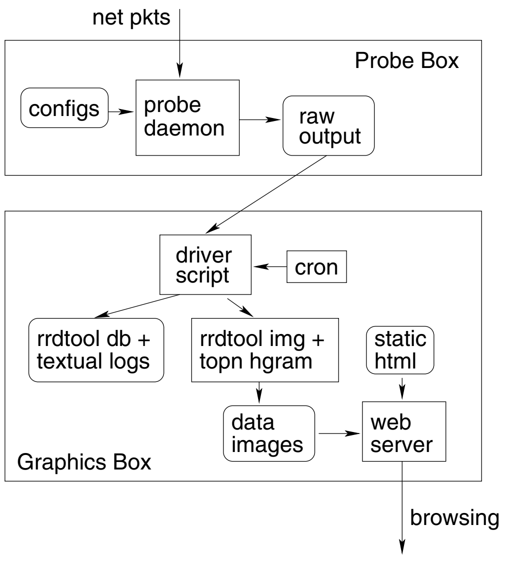


Figure 2: Ourmon software architecture

then reads packets from the command-line specified network device. It outputs various files depending upon configured filters including the primary *mon.lite* file and other optional secondary tuple files. The graphics engine script is responsible for transferring the raw probe data files to the display machine (if necessary), and invokes analysis tools to produce graphics for the web interface and also produces log information. The graphics engine script is invoked by cron once per minute and actually runs to completion twice per minute – hence the sample period is thirty seconds. The graphics engine places the graphics and some analyzed report data in the web output directory thus refreshing the data encapsulated by the web pages installed at configuration time. The script here also produces logging information not available on the web, which may be looked at for layer analysis. The user interface for the resulting display of data is the user's web browser of choice.

The Ourmon web interface is quite simple. Ourmon can be observed in operation in the PSU DMZ by visiting `http://ourmon.cat.pdx.edu/ourmon` with a graphical web browser. From there, a variety of RRDTOOL and other filter outputs are available. More reports can be accessed simply by clicking on the appropriate links. The top-level web page in general shows current RRDTOOL pictures, current histogram pictures for top-N data and also has links to some kinds of sample period or summarized daily reports. The top page may be viewed as a directory for the various RRDTOOL filters, and other kinds of filters. This is because it is intended that there should be at least one current picture for every filter type on the top level page. The top picture may in turn lead to a link that provides more filter details on lower level pages. If we focus on RRDTOOL filters, each filter has its current graph for today on the top-level page with ”now” on the right-hand edge of the picture. That graph in turn is a link that leads to a second-level page that has graphs for the current day (roughly), the current week, the current month, and the current year. Thus each RRDTOOL filter provides a year’s worth of baselined data (this is a typical RRDTOOL toolset feature and is not a noteworthy Ourmon feature). Top-N data is similar. The current picture for the top 10 tuples is shown on the top page, and that picture in turn is a link to a second-level page that provides supplemental tuples. Because the current Ourmon front page is quite large, it is not practical to provide a screenshot. However, the displays shown in this paper give a good indication of the kinds of graphical display available.

2.2 Configuration and Use

The Ourmon probe process uses the BPF in two ways. The BPF library is used to extract packets from the kernel BPF buffer system. Ourmon also allows the administrator to evaluate multiple concurrent BPF expressions in user mode. In the probe configuration file, a user can group logically related BPF expressions in a BPF filter-set. Each expression in the set can be graphed as a separate line in a shared RRDTOOL strip chart graph in the back-end. Such filter-sets have a name, provided by the user in the front-end config. The back-end uses the filter name to synthesize an RRDTOOL database expression, and to create runtime graphics for any new filter-set. We provide a collection of BPF filter-sets with our default install, many of which perform useful network anomaly detection tasks. It is also easy to configure new graphical outputs for Ourmon data. Graphical outputs are described using the ”tcpdump” expression language as found in libpcap [19]. Tcpdump and other common network tools also use these filters. After creating a new probe configuration filter, the user must give it a unique name. A small amount of HTML, copied from our template files, is usually sufficient to glue the new filter to the supplied main page. We hope to further automate this process in a future release. Thus Ourmon may be easily extended by a user with a new BPF-based set of expressions.

As one example, Figure 3 shows a simplified probe configuration for one BPF filter-set. This filter-set groups the performance of five application services together and uses one BPF expression each for ssh, combined P2P protocols, web, FTP, and email. Probe output is not intended for human consumption, but is useful for debugging. The probe output for the filter above over a snapshot period might look like this:

```
bpf "ports" "ssh" "tcp port 22"
bpf-next "p2p" "tcp port 1241 or
     tcp port 6881"
bpf-next "web" "tcp port 80 or
     tcp port 443"
bpf-next "ftp" "tcp port 20 or
     tcp port 21"
bpf-next "email" "tcp port 25"
```

Figure 3: Ourmon probe configuration

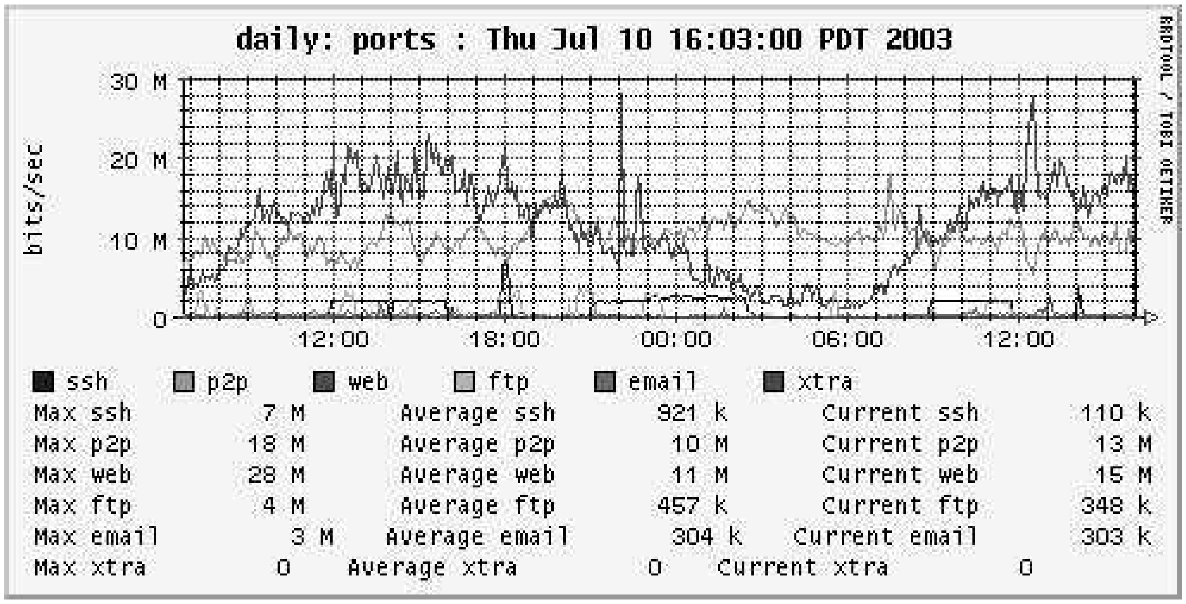
```
bpf:ports:5:ssh:254153:p2p:19371519:
 web:41028782:ftp:32941:email:1157835
```

Thus ssh/p2p/web/ftp/email byte counts will all appear in the same RRDTOOL graph as in Figure 4.

The filter configuration allows the user to name the composite filter-set graph “ports”. This is accomplished using the “bpf” configuration tag. This tag provides a line label and initial expression for the graph. The configuration tag “bpf-next” adds another BPF expression to the graph. The graph may be terminated one of several ways, including a new “bpf” tag, which starts a new graph. Overall, five separate user-mode BPF configuration expressions like “tcp port 22” are mapped to appropriate line labels (“ssh”) in the same graph. (This graph is taken from the PSU DMZ and shows web traffic and P2P traffic as the biggest bandwidth consumers.) The probe executes the user-mode BPF runtime expressions on the incoming packet stream from the packet tap, counting matching bytes or packets. At the sample period timeout, it outputs the *mon.lite* file. In this case, the file includes the name of the filter-set and line label / byte count tuples for each BPF expression. Note that multiple BPF filter-sets are possible. Thus many separate BPF expressions can be executed in the probe application. At the time of writing, the current PSU DMZ probe software is running around 80 BPF expressions in twenty filter-sets.

Ourmon also supports a small set of “hardwired” filters programmed in C and turned on via special filter names in the configuration file. For example, a hardwired filter counts packets according to layer 2 unicast, multicast, or broadcast destination address types. One very important filter called the *packet capture* filter includes statistics on dropped and counted packets provided directly from the BPF kernel code. The packet capture filter is fundamental. It is used to determine when the kernel BPF mechanism plus application mix is overloaded in our testing. Typical front-end output in the *mon.lite* file for that filter and the layer 2 packet address type filter might look like this:

```
pkts: caught 53420 drops: 0
fixed_cast: mcast: 2337215:
 unicast: 15691896: bcast: 0:
```



```
Max ssh        7 M     Average ssh      921 k    Current ssh     110 k
Max p2p       18 M     Average p2p       10 M    Current p2p      13 M
Max web       28 M     Average web       11 M    Current web      15 M
Max ftp        4 M     Average ftp      457 k    Current ftp     348 k
Max email        3 M   Average email      304 k  Current email   303 k
Max xtra         0     Average xtra        0     Current xtra      0
```

Figure 4: A BPF filter-set graph showing application byte counts

The *packet capture* filter ("pkts") output shows that the packet tap during the last sample period caught 53240 packets and dropped none. In Figure 5 we show an example back-end graph for this filter. The upper line in the figure indicates captured packets: the lower line indicates drops. This graph is from our DMZ during the day of a Slammer re-infection. It can be seen that the Ourmon probe, at the time running on a Pentium-3, has caught the attack even though many packets have been dropped. This is a real-world example of small packets causing a monitor system to underperform. We have also seen distributed TCP SYN attacks cause the same phenomenon.

The third and last filter class in Ourmon are top-N based filters. Top-N filters produce a sublist of limited set size: the top-N elements of the entire sorted list. The list is characterized by a tuple that includes a key and a set of integer counters. In some cases a nested list of tuples might include sampled destination ports or other data. Tuple keys may be as simple as an IP source address. This has proven to be a very profitable focus for network summarization and anomaly detection. An IP flow from the top-N flow filter can also be a tuple key. Tuple-based filters currently include:

1. The traditional top-N talker flow filter that tells us the top IP, TCP, UDP, and ICMP flows. We view this filter as typical of its class: in this paper we focus only on measurements related to this filter.
2. A top-N port monitor that tells us which TCP and UDP ports are the most used.
3. A top-N TCP SYN monitor that is quite useful in anomaly detection. It includes a basic top talker graph to tell us which IP hosts are sending the most SYNS. It also includes several important anomaly detection functions including a *port signature report* that reveals "noisy" hosts with a set of sampled destination ports. These hosts are typically P2P systems or scanners or hosts infested with worms. An RRDTOOL-based worm graph gives us a count of such noisy hosts at any one time. This SYN tuple filter is the focus of much ongoing research.
4. A top-N scanning monitor that tells us which hosts are doing IP destination scanning, and which hosts are doing L4 (TCP and UDP) destination port scanning.
5. A top-N ICMP error monitor that tells us which hosts are generating the most ICMP errors and as a side effect, which systems have created the most ICMP errors with UDP packets.

When configured to use the top-N flow filter, the probe builds up a hash-sorted list of IP flows over the sample period and writes the top-N, say 10 to 100, IP flows to the main output file. It also writes subsets of the IP flow including TCP, UDP, and ICMP flows. The graphics-engine takes this information and produces graphical histograms and text report summaries hourly. The key for this tuple type is a flow. A flow is a five-tuple consisting of IP source, IP destination, IP next protocol, L4 source port, and L4 destination port. See Figure 6 for an example of back-end graphics for the top-N report: we show a DOS attack on a local IT administrator's host machine. The top six flows in the graph con-

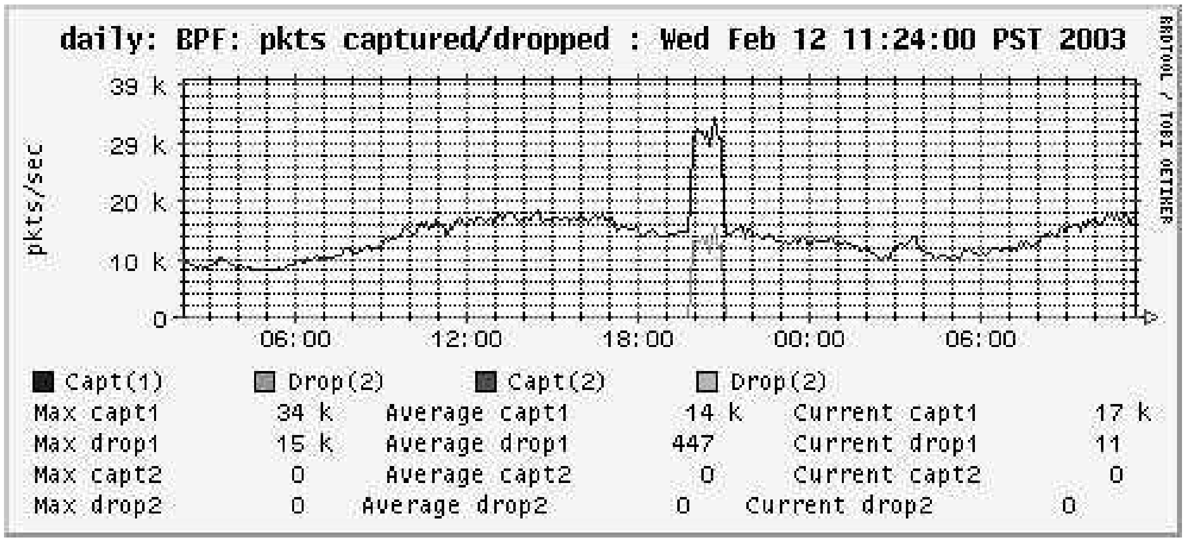


Figure 5: The packet capture filter graph showing counts and drops during a slammer attack

stitute the attack. The attack packets were launched over Internet2 using a spoofed IP source address and unfortunately clogged our Internet1 connection (a DS-3 at the time). Multiple UDP flows, each around 1.5 Mb/s, are shown. This picture is of historic significance to us as it was discovered during a “demo” on the first day that Ourmon was ever deployed in PSU’s network. This result emphasized to us the fact that Ourmon is not just a networking monitoring system, but also an anomaly detection system.

In summary, the front-end has three kinds of filters: hardwired C filters, user configurable BPF filter-sets, and top-N tuples including a flow filter. We are interested in the execution cost of each of these three kinds of filters. Of our tuple types, we have chosen the top-N flow filter as representative of its class. It is also the first filter type developed in its class so we thus have the most experience with it. A user may program any number of BPF filter-sets and this complicated the analysis somewhat.

The *packet capture* filter is especially important, as it serves to tell us when we are losing packets. We can view this as an important indicator that the combined kernel and probe application system is in failure mode. An important cause of failure is too much work done at the application layer, causing the application to fail to read buffered kernel packets in a timely manner.

3 Experimental Setup

The hardware used in our testing consists of three pieces of equipment:

1. An IXIA 1600 chassis-based packet generator with a two port GigE line card. One port sends packets and the other port receives packets.
2. A Packet Engines line speed GigE switch. Three ports on the switch are used: one for the IXIA send port, one for the IXIA receive port, and a third port connected to the UNIX host for mirroring the IXIA flow.
3. A 1.7 GHz AMD 2000 computer system. The AMD processor is roughly comparable to a 2GHz Intel Pentium 4 processor. The system motherboard is a Tyan Tiger MPX S2466N-4M. The motherboard has two 64-bit PCI slots. We use a SysKonnect SK-9843 SX GigE card in one of the slots.

Software used includes Ourmon 2.0 and 2.4, (2.4 at the time of writing), along with the 0.7.2 libpcap [19] library. The host operating system is FreeBSD 4.9, running only the Ourmon front-end probe.

We set up the IXIA to send either minimum-sized packets or maximum-sized Ethernet packets. One port on the IXIA sent packets through the switch to the other IXIA port. All packets were UDP packets.

The IXIA allows the user to select an arbitrary packet sending rate up to the maximum possible rate. It can also auto-increment IP destination addresses. We used this feature as an additional test against the top-N filter.

According to Peterson [7] the maximum and minimum theoretical packet rates for GigE are as shown in Table 1. We used these values as a measurement baseline. We observed that the IXIA 1600 can indeed generate packets at nearly 100% of this rate for both maximum and minimum-sized packets. We used these numbers to make sure that our Ethernet switch did not drop packets. We hooked both IXIA GigE ports up directly to the switch and sent packets from one IXIA port to another.

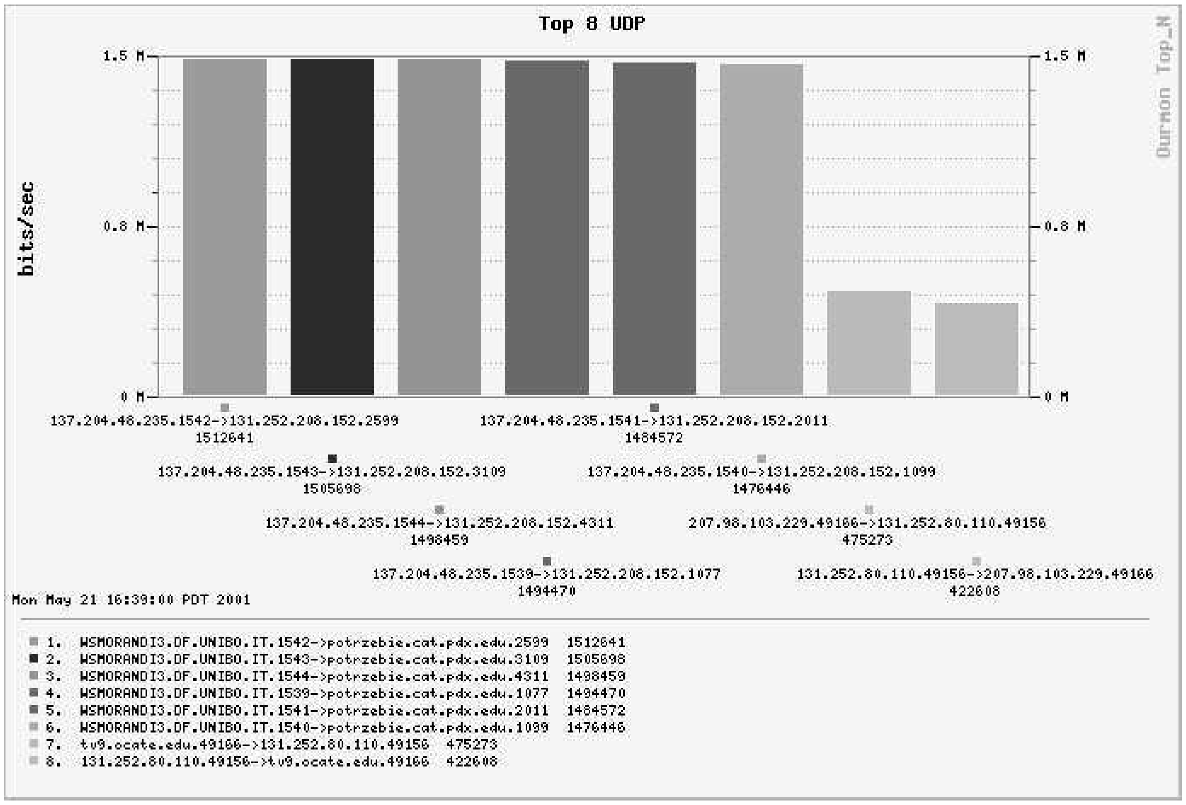


Figure 6: Top-N UDP flow histogram showing a DOS attack

Table 1: GigE rates

	B/pkt	pkt/s
min	64	1488000
max	1518	81300

The IXIA's built-in counters at the receive port reported the same packet counts as at the send port.

The test methodology involves setting up a UNIX host with a driver script and some set of Ourmon filters. The front-end probe is started, and the IXIA is configured to send min or max packets at some fraction of the maximal rate. Ourmon is configured with some combination of hardwired, user-mode BPF, and the top-N filter as desired. The test flows are then started on the IXIA, and the results observed using the *mon.lite* output file.

The test script is the Bash Shell script shown in Figure 7. The FreeBSD *sysctl(8)* command is used to set the kernel BPF buffer size. This is because recent versions of the pcap(3) library on FreeBSD will automatically size the buffer to be used by the client application to match the kernel buffer size. It should be noted that the traditional size of the kernel BPF buffer is typically small (a few KB/s), as it was originally intended for the tcpdump sniffer. The parameters to the Ourmon probe program tell it to take input from a local configuration

```
#!/bin/sh
BSIZE=1048576
sysctl -w debug.bpf_bufsize=$BSIZE
sysctl -w debug.bpf_maxbufsize=$BSIZE
./ourmon -a 5 -I sk0 -m /dev/tty \
  -f ./ourmon.conf
```

Figure 7: Test script

file, to dump the output information to the screen every five seconds, and to use the SysKonnect card as the input interface.

Tests were run using either maximum-sized or minimum-sized packets. If we dropped packets, we attempted in every case to eliminate packet drops by increasing the kernel BPF buffer size (BSIZE above). If that failed, we then reduced the IXIA's send rate until all packets were transmitted.

For testing, we identified five interesting categories of Ourmon filters and constructed filter tests for these categories.

null: The packet capture filter is on by default and is the only filter used.
hard: The hardwired C filters as a group.
bpf: BPF filters as one or more filter-sets.
top-n: The top-N flow filter mechanism.

Table 2: Maximum Packet Tests

test	BPF sets	top-n flows	BPF min (KB/s)	drop rate
null filter			128	0%
hardwired			128	0%
top-n		1000	128	0%
top-n		10000	XXX	80%
BPF	1		128	0%
BPF	4		128	0%
BPF	8		128	20%
BPF	8		7168	0%
test config	1	1000	7168	0%

combo: A simple combination of all filters.

The null filter tells us whether or not the BPF in the kernel was losing packets, as its count/drop information is taken from the operating system. The hard, bpf, and top-N filter categories were tested individually in order to determine if the filter type itself had an impact on the overall performance. The six hardwired C filters available at the time of testing were used in the tests. The bpf tests were based on a filter-set that had 4 simple expressions in it. The individual BPF expressions were configured to capture TCP ports that could not match the output of the IXIA (UDP packets). It seemed reasonable for BPF expressions to always fail to match.

Repeatedly testing the top-N filter with the same IP flow would yield no new information. Therefore, for the top-N test we used a rolling IP destination setup where each subsequent UDP packet within a set of 1000 or 10000 had a different IP destination. This could be said to be a rough simulation of the Slammer worm, with its variation in IP destinations.

4 Test Results

Test results fall into two basic categories, which are reported separately: tests with maximum-sized packets, and tests with minimum-sized packets.

4.1 Maximum Packets

In this set of tests, packets were always 1518 bytes, the normal maximum MTU for Ethernet packets. (This works out to a 986 Mb/s flow of UDP packets). Tests included the null, hardwired, top-N (with different destination flow counts), bpf, and combo tests.

The test results are summarized in Table 2.

The flow rate was set to maximum. The drop rate therefore shows packets lost at GigE speeds. In the null case, the configuration *almost* worked with the typical BSD default BPF buffer size of 4 KB/s. However, some packets were lost at a 30 second interval. This may have had something to do with an operating system timer. Increasing the kernel BPF buffer size to 128 KB/s resulted in perfect transmission, even after adding in the hardwired filters.

The top-N flow filter worked with no loss at 1000 flows and completely failed at 10000 flows. Larger BPF buffers did not help (shown as XXX in the table). This is the most significant failure case with maximum-sized packets. Decreasing the IXIA flow rate to 45 Mb/s resulted in perfect transmission. For the bpf tests, we increased the number of filters to 8 sets (32 BPF expressions) before running into some loss. At that point, we increased the kernel BPF buffer size. We found that a very large buffer of 7 MB/s could indeed get us back to lossless transmission. With the combo configuration (hard + top-n + 1 bpf set) we did not experience any loss. Note however that we used only 1000 flows with the top-N filter.

Table 3: Minimum Packets and Null Filter

BPF buff (KB/s)	drop thresh (Mb/s)
32	53.33
128	68.52
256	76.19
512	76.19

4.2 Minimum Packets

Attempts to capture maximum-rate flows of minimum-sized packets (64 bytes) uncovered serious problems. We therefore report our results as a series of small experiments. Each experiment focuses on a different test domain.

4.2.1 Null Filter Only

With the null filter, we are not doing any significant application work. Consequently, this test determines whether the kernel driver and buffer subsystem plus the application read can actually capture packets. It was not always possible to capture all packets even in the null filter case. Instead we attempted to determine the effect of the kernel BPF buffer size on drop rates as shown in table 3.

A buffer size of 256 KB/s appears optimal. At this size the system begins to drop packets at 76 Mb/s. Larger kernel buffers do not improve the result. Of course the most important aspect of this test is that we cannot capture more than around 10% of the GigE stream without loss. (Note that packet overhead for minimum packets results in a maximum data flow of around 760 Mb/s.)

Table 4: Hardwired and BPF Tests

test	BPF sets	flow (Mb/s)	drops
hardwired		76	0%
BPF	1	68	0%
BPF	2	53	0%

Table 5: Minimum Packets—top-N Tests

flows	drops	buffer (KB/s)	flow (Mb/s)
1	0%	256	76
100	1%	256	76
1000	25%	256	76
1000	0%	256	45
10000	50%	*	*

4.2.2 Individual Filter Types

Having determined baseline drop rates using the null filter, we could now proceed to measure the impact of other filter types. In the bpf filter-set tests, we tried both one and two filter-set configurations. In the top-N filter test, we varied the number of simultaneous flows. Table 4 shows the results for the hard and bpf tests. Table 5 shows the results for the top-N tests.

Hardwired filters appear to have no impact on performance. The bpf filters have some performance impact, visible even at a modest 76 Mb/s transfer rate. At this transfer rate, 1000 unique flows is stressful for the top-N filter. However reducing the flow rate to 45 Mb/s allows the filter to keep up with the data. 10,000 unique flows cannot be handled with any kernel buffer size at any measured transfer rate.

4.2.3 Combination filtering

In this experiment we measure the combo filtering previously discussed. Here we vary only the flow rate, holding the buffer size constant at 256 KB/s and the number of flows constant at 1000. Table 6 shows the results.

We see that we must reduce the flow rate to roughly one-half maximum in order to prevent drops. This is probably because of the impact of 1000 flows on the top-N filter. The filters here are in truth fairly minimal, as there is only one BPF filter-set. In reality one would want more filter-sets to get better traffic information. The bottom line is that we must reduce the flow rate to 38 Mb/s for even a modest amount of work to be performed without packet loss. Not only are small packets hard to deal with even for bare-bones applications that do no real work in processing them, but real levels of work will likely reduce the amount of processing power to very small throughput rates.

Table 6: Minimum Packets—All Filter Types

flow (Mb/s)	drops
76	44%
68	37%
53	18%
45	03%
38	0%

5 Mitigation

The poor performance of the Ourmon probe on even modest flows of small packets was of obvious concern. In the real world in PSU's DMZ we feel that Ourmon is useful as an anomaly detector even under conditions of severe packet loss. Indeed, such a loss constitutes an anomaly indicator in its own right. Nonetheless, the exhibited performance of Ourmon in the lab on minimum-sized packets was unexpectedly poor. This poor performance was a threat to some of the conclusions reached in real-world use. Several strategies were thus pursued in improving probe efficiency.

The top-N flow filter has been both one of Ourmon's most useful tools and one of its least performant. It was observed that the hashing/caching strategies and data structures used in the initial implementation of this feature could be vastly improved. In optimizing the flow filter code, our main strategy was aimed at improving the runtime hashing mechanism. A simple but key improvement was in choosing an appropriate size for the hash buffer. The hash function and hashing strategy were also changed to directly address the problem of hashing a traditional flow tuple. Other efficiency improvements included inlining the basic search and insert functions and the hash function itself.

The user-level interpreted BPF filter performance was also a cause for some concern. Our real-world PSU DMZ probe has recently been seeing peaks around 40000 pkt/s, and has had 80 BPF expressions in its configuration. The top-N flow filter used to be the main bottleneck. Over a number of years we have increased the number of BPF expressions used in our DMZ and have made the BPF sub-system the new contender for that honor. Our idea for improving packet filtering performance was not a particularly clever one, but was quite effective.

We have created a simple runtime facility, CBPF, for hand-coding commonly used BPF expressions as C subroutines. The probe configuration file may refer both to BPF expressions and to CBPF functions. One may thus optionally replace a commonly-used interpreted BPF expression with a call to a C function via a function jump table. For example, roughly half of the BPF expressions we are using in our DMZ simply watch subnets.

Consider this sample BPF subnet-recognizing expression from a configuration file:

```
bpf "subnets" "subnet 1"
  "net 192.168.1.0/24"
```

The CBPF replacement for this is simple:

```
cbpf "net" "subnets" "subnet 1"
  "192.168.1.0/24"
```

This is not terribly sophisticated engineering, but it gets the job done. The BPF interpreter remains useful, as its expression language is versatile. In general, however, long or commonly-used BPF expressions can be optimized; most of the filtering in the PSU Ourmon configuration is now being performed by CBPF.

Ourmon has been modified in order to analyze the effect of our optimizations. Instead of taking data from the network, we can capture real packets in the PSU DMZ with tcpdump. We can then take the dump data and feed it to the Ourmon probe. This allows us to use gprof profiling to determine relative speed improvements for code changes.

We recently conducted an experiment using 10 million packets— roughly 1 GB of data—from our DMZ. We compared the difference between a not-yet-released version of Ourmon and the older Ourmon version 2.2. Ourmon 2.2 lacked top-N optimizations and CBPF support. (The currently released version 2.4 of Ourmon has the top-N optimizations in it. However, while CBPF will be available in the next release, it is not in 2.4.) We compared the performance of CBPF expressions versus BPF for filtering; we also analyzed the performance of old versus new top-N code. The performance improvements were gratifying. CBPF expressions proved to be roughly 10 times faster, with the performance improvement depending on the complexity of the expression. The improved top-N facility was roughly 30 times faster.

As expected, these mitigations resulted in greatly improved performance when fielded in the PSU DMZ. As with most large intranets, the traffic volume in the PSU DMZ is quite cyclic. When Ourmon used interpreted BPF expressions for all filtering, it routinely dropped many packets during peak usage times. Installing Ourmon with CBPF essentially eliminated this problem. Figure 8 illustrates this phenomenon. The upper line in the figure indicates weekly traffic volume. The lower line indicates the number of dropped packets during the time period when Ourmon CBPF was introduced. The date the optimization is installed (week 5) is quite apparent from the graph.

Other mitigations are highly desirable, but require much more effort. Such schemes might include: placing the probe in the kernel; various forms of small-scale threaded parallelism done either at the application or kernel level on a SMP platform; and putting the probe into a network card, possibly using something like the Intel IXP [2] network processor which uses embedded parallel processors. We have experimented to some extent with all three of these options. In the near term we intend to pursue the more portable application-level parallelism option, although we do not rule out an IXP port.

6 Related Work

Ourmon touches on a number of areas of computing research and practice. Minimally, any discussion of related work must include comparison with existing tools, background related to the BPF, and work on the problem of processing small packets.

In a broad sense Ourmon could be compared to simple sniffers like tcpdump. Ourmon's goal is somewhat different, however. Sniffers focus on displaying a serial list of individual packet contents based on one expression. Ourmon, on the other hand, tries to summarize many packets and give a parallel view of network traffic via many concurrent BPF expressions.

The measurement system closest to Ourmon is probably Ntop [5]. Ntop and Ourmon are both open source. Ntop is a program that can be said to be vertically integrated. It combines the probe and graphics engine functionality of Ourmon plus a web server into one program. Ntop could be said to have a desktop orientation. It derived its name from its origins as a network version of the UNIX top program. Ourmon was designed more along the lines of the traditional distributed SNMP RMON probe from which it derives its name. Ourmon easily decomposes into a two-CPU system design separating capture and display. Ourmon also has a network-statistical feature set more in keeping with recent cultural trends in network engineering. For example, Ourmon relies heavily on RRDTOOL-based graphics. RRDTOOL is also used by other popular network management tools like Cricket [21, 20]. Ntop has a much better graphical user interface than Ourmon—perhaps there may be room there for future joint effort. Close examination of the feature sets of Ourmon and Ntop shows significant differences.

Ourmon might also be compared to closed-source commercial tools like SNMP RMON II. One of the original design goals of Ourmon was to provide a rough open source equivalent of SNMP RMON. This notion includes the fundamental probe versus graphics engine two-CPU design. On the other hand, Ourmon deliberately used a human-debuggable TCP-based ASCII tuple format in an effort to avoid the complexity inherent in the implementation and use of the ASN.1 remote procedure call notation. Ourmon makes no attempt to use a

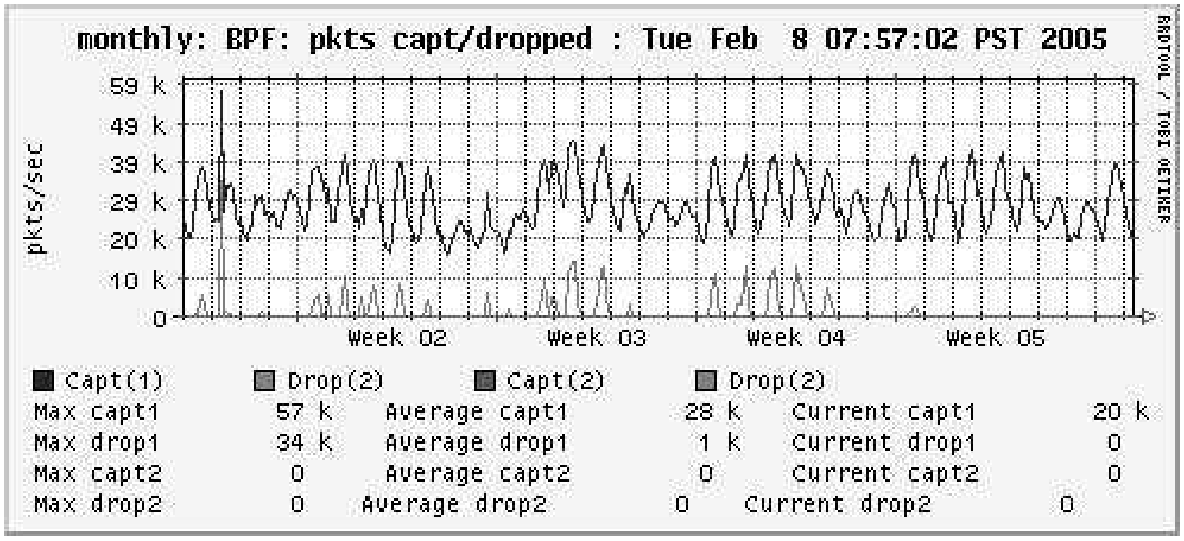


Figure 8: Packet losses without/with CBPF

standardized network protocol to join the probe and the graphics engine. On one hand, this inhibits interoperability of these components. On the other hand, it allows easy and unconstrained changes in component communication with each new version.

In a similar vein, Ourmon could also be compared to Cisco's NetFlow[1]. We should point out that there exist some open source NetFlow probes. However typically one encounters a NetFlow probe in a more expensive commercial router or switch. At least one open source NetFlow collector, NEye [13], is available; the reporting functionality of NEye does not yet appear to be comparable with that of the closed-source Cisco collector. NetFlow aggregates IP flows along the traditional lines of an IP flow tuple: IP src, IP dst, protocol, L4 src, L4 dst ports. Ourmon's new tuples, such as its TCP SYN tuple, tend to be targeted toward specific features interesting for anomaly detection. Commonly, these are statistics about a single IP source. Our not-yet-released experimental version of Ourmon includes statistical analysis based on Layer 7 payloads. This sort of analysis is currently impossible to perform using NetFlow, and it is difficult to see how small changes to the NetFlow architecture could accomodate it.

From the intrusion detection point of view, Ourmon and Ntop are somewhat similar. They are lightweight tools that show anomalous behavior via graphs. In contrast, an Intrusion Detection System (IDS) tool like Snort does signature-based analysis on every packet. One could argue that Ourmon is lightweight compared to Snort. Ourmon looks principally at the layer 1–4 network headers and almost entirely ignores the data payload. It is thus reasonable to expect that Snort's processing will be impacted even more than Ourmon's by flows consisting of high volumes of small packets. As Ourmon's analysis of data payloads increases, this difference may decrease. On the other hand, future versions of Snort might also be expected to increase their level of analysis. The bottom-line difference is simply that Ourmon is an anomaly detector and analysis tool, while Snort is primarily a signature-based IDS.

Other researchers have studied the problem of capturing high-volume flows of small packets. For example, Mogul and Ramakrishnan [11] describe the phenomenon of *receive livelock*, in which the network device driver bottom-half runs to the exclusion of higher-level code that processes received packets. They present improved operating system scheduling algorithms that can lead to fair event scheduling, with the result that receive interrupts cannot freeze out all other operating system events.

One must consider that there is not a lot of time to process packets. A maximal packet flow of about 1.5 million small packets per second works out to approximately 700 nanoseconds per packet! Some sophisticated approach, such as improving the individual compute performance of various filter mechanisms or applying parallelism, is needed to attain adequate performance. A recent IDS [8] contains an interesting parallel hardware engine based on a flow slicing technique. This hardware reportedly improves Snort's performance under high packet loads. However constructing such a system in such a way that it effectively uses parallelism and yet remains cost-effective is a challenge.

Recently, researchers have been working on enhancements to the BPF with the goal of improving BPF performance. For example, the xPF system [6] expands the BPF to a general purpose computing machine by allowing backward branches. This provides the opportunity to enhance BPF performance by running filters entirely in-kernel. The BPF+ system [3] optimizes BPF performance using both machine-code compilation and various optimization techniques. Our CBPF, while much less sophisticated, echoes the goals and general method of this latter work.

7 Analysis

Our experimental work leads to some interesting recommendations and observations:

1. The default FreeBSD BPF buffer size of a few KB/s was chosen for tcpdump on older, slower networks: this size is inadequate for more thoroughly monitoring a modern network. We suggest that modern UNIX/Linux kernels adopt a larger default buffer of at least 256 KB/s. This size should not unduly burden modern systems, and should improve the performance of most network monitoring tools. Network administrators should understand that a multi-megabyte buffer on the order of 8 MB/s may be needed for a full network monitoring system such as Ourmon. Larger buffers should improve the performance of the Linux packet socket, should minimize the loss of large packets by network sniffing applications such as Snort.
2. Our BPF filters seem to have a kernel buffer cost associated with them. Our results suggest that there is a relationship between the amount of kernel buffer space needed to mask filter latency and the number of BPFs used in our application. Our tests seem to imply that the BPF mechanism is less costly than the top-N filter. However the BPF mechanism can have any number of expressions, and the expressions themselves can vary in complexity. It is thus hard to compare the BPF filter mechanism to the top-N filter mechanism in terms of compute power.
3. The real computation problem for the top-N system is that it is driven to extremis under attack attempting to cope with random IP source and/or destination IP addresses. The hash-based top-N algorithm will first search for the given flow ID, and then perform an insert if it fails to find the flow. Consequently random flows always cause an insert. This leads to an interesting research question: How can we deal with boundary conditions caused by random IP addresses without unduly impacting efficiency mechanisms meant for normal bursty flows?
4. Our 2 GHz Pentium-4 class computer cannot capture more than 10% of the minimum-sized packet flow. Worse, if the computer is expected to perform actual application-level work using the data, the fraction of packets we capture without loss falls below 5%. Small packets mean big trouble.

This last item deserves extended discussion. Consider an IDS system such as Snort. A signature-based IDS system wants to run an arbitrary number of signatures over both the packet headers and the packet data, and may choose to store its measurement results in a database. Clearly per-packet processing times become quite large in this scenario.

Now consider the security principle known as *weakest link*. For example, Bruce Schneier writes [18]: “Security is a chain. It’s only as secure as the weakest link.” An IDS system incurs a significant risk when it drops a single packet. The dropped packet may be the one with the Slammer worm that will infect an internal host. Worse, a set of coordinated systems might launch a distributed DOS attack against an IDS monitor, first blinding it with small packets and then sneaking a one-packet worm payload past it. Packet capture for small packets at high rates is an important open security problem.

It should also be pointed out that the natural evolution of any network monitoring tool is toward more functionality at the expense of more work. For example, we have recently added many new kinds of list tuples to Ourmon in addition to our original top-N flow tuple. Many of these new tuples are useful for both network management and anomaly detection, We are also starting to expand our packet analysis work to Layer 7. In the extreme case, Snort and Ourmon could be combined.

In addition use of the Internet is always growing. When Ourmon was originally placed a few years ago in the PSU DMZ, we saw 20 K pkt/s at peak times. Now we see peaks of 40 K pkt/s. When both new tool features and packet counts grow, and these factors are combined with the possibility of large distributed attacks, it is fair to say that the computation overhead problem is non-trivial.

A related open research question: other than by trial and error, how does one determine if a measuring system is powerful enough to fit the needs of a certain network configuration? From real-world experience in our DMZ, we know that a 3 GHz P4 is challenged by 40 K pkt/s flowing from a moderately-loaded 100 Mb/s network connection. Such a system, however, may work well with a 10 Mb/s Internet connection. An administrator faced with 500 Mb/s peaks, or even an OC-3 (155 Mb/s) faces a difficult problem in specifying a hardware environment adequate for use with a signature-based IDS such as Snort, a network monitor and anomaly detector such as Ourmon, or even a firewall.

Our future work will be aimed in several directions. We think that the small packet problem must be addressed. We plan to further investigate various parallel architecture notions. In particular, we are hoping that a threaded SMP solution will prove sufficient. This could lead to an open source multi-platform (BSD/Linux) probe.

Although we have not provided many details of our recent anomaly detection work in this paper, we believe that our recent work in the last year in that area has been promising. (See section 8 below for more technical information.) We intend to steer Ourmon further in this direction. For example, we are beginning to investigate lightweight Layer 7 payload scanning statistics that may help us find peer-to-peer applications as well as logical networks composed of IRC bots ("botnets"). We are also studying the effects of various statistical schemes for cheap classification of attackers and peer-to-peer applications.

We are in the process of bringing up an automated trigger-based packet capture facility. This facility allows us to specify a threshold for any BPF expression and for some of the top-N style graphs. Packets are captured during peak times in which a threshold is exceeded, and stored in tcpdump capture files for later analysis. This facility should prove useful in characterizing and analyzing anomalies reported by Ourmon.

8 Conclusion

Ourmon is a novel tool for network monitoring aimed principally at anomaly detection and analysis. Experiments measuring the performance of both the underlying kernel BPF filter system and the Ourmon front-end filter systems have led to dramatic improvements in Ourmon's performance. Ourmon's probe uses the BPF and its CBPF in a flexible fashion, allowing the user to group a small set of related filter expressions into a single RRDTOOL graph. Ourmon also provides various graphs and reports about tuple lists keyed by flow IDs, IP source addresses, and L4 ports, which are intended to summarize statistically significant network events.

Ourmon fills an important niche in the open source network monitoring toolset. It also points up some fundamental performance issues in network monitoring. Addressing these issues in Ourmon has led, and should continue to lead, to improvements in general networking monitoring performance and a better understanding of these performance issues.

Availability

Ourmon is freely available at `http://ourmon.cat.pdx.edu/ourmon` under the BSD License. More technical information may be found for the current release at `http://ourmon.cat.pdx.edu/ourmon/info.html`.

9 Acknowledgements

This work would not have been possible without the generous donation from the IXIA Corporation of an IXIA 1600 tester. We also wish to thank our graduate students Subha Singaram and Gayathri Nagabhushan for their help with the measurement project.

References

[1] Cisco Systems. Cisco CNS NetFlow Collection Engine. `http://www.cisco.com/en/US/products/sw/netmgtsw/ps1964/products_user_guide_chapter09186a00801ed569.html`, April 2004.

[2] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Worich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, August 2002.

[3] A. Begel, S. McCanne, S. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. *Proceedings of ACM SIGCOMM*. September 1999.

[4] CERT Advisory CA-2003-04 MS-SQL Server Worm. `http://www.cert.org/advisories/CA-2003-04.html`, November 2003.

[5] L. Deri and S. Suin. Practical Network Security: Experiences with ntop, *IEEE Communications* Magazine, May 2000.

[6] S. Ioannidis, K. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: Packet Filtering for Low-Cost Network Monitoring. In *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (MPSR)*, May 2002.

[7] Karlin, Scott, Peterson, Larry, Maximum Packet Rates for Full-Duplex Ethernet, Technical Report TR-645-02, Department of Computer Science, Princeton University, Feb. 2002.

[8] C. Kruegel, F. Valeur, G Vignka, R. Kemmerer. Stateful Intrusion Detection in High-Speed Networks. In *Proceedings IEEE Symposium Security and Privacy*, IEEE Computer Society Press, Calif. 2002.

[9] Leffler, et. al., *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison-Wesley, 1989

[10] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter 1993 USENIX Conference*, San Diego, January 1993.

[11] J.C. Mogul and K.K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. In *ACM Transactions on Computer Systems*, 15(3):217-252, August 1997.

[12] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, N. Weaver. The Spread of the Sapphire/Slammer Worm. `http://www.cs.berkeley.edu/~nweaver/sapphire`. 2003.

[13] NEye, an Open Source Netflow Collector. `http://neye.unsupported.info`. February 2005.

[14] Ntop distribution page. `http://sourceforge.net/projects/ntop`. March 2004.

[15] Ourmon web page. `http://ourmon.cat.pdx.edu/ourmon`, March 2004.

[16] M. Roesch. Snort—Lightweight Intrusion Detection for Networks. *In Proceedings of the USENIX LISA '99 Conference*, November 1999.

[17] RRDTOOL web page. `http://people.ee.ethz.ch/~oetiker/webtools/rrdtool`. March 2004.

[18] B. Schneier. *Secrets and Lies*. p. xii. Wiley Computer Publishing. 2000.

[19] Tcpdump/libpcap home page. `http://www.tcpdump.org`, March 2004.

[20] Cricket home page. `http://cricket.sourceforge.net`, March 2004.

[21] J. Allen. Driving by the Rear-View Mirror: Managing a Network with Cricket. *In Proceedings of the USENIX First Conference on Network Administration '99 Conference*, April 1999.

[22] Waldbusser, S. Remote Network Monitoring Management Information Base Version 2. IETF. RFC 2021, January 1997.

Brooery: A Graphical Environment for Analysis of Security-Relevant Network Activity

Christian Kreibich
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
christian.kreibich@cl.cam.ac.uk

Abstract

We present the design and implementation of the Brooery, a system for graphical analysis of network activity reported by instances of the Bro intrusion detection system. It supports multiple input streams and provides a web-based graphical user interface to allow the user to analyze the reported activity. The Brooery understands activity at different abstraction levels, allows for quick drill-down searches by focusing on contextuality when moving through the history of events, and provides user-friendly and semantically strong hierarchical filtering to reduce the amount of information presented.

1 Introduction

In recent years, network monitoring has become a widely adopted practice for practically every organization interested in understanding the activity on its networks. Besides other reasons, this is mostly done to improve security: intrusion detection systems (IDSs) are nowadays widely deployed in order to help analysts focus their attention on critical events that otherwise might have been missed in the vast amount of activity.

While these systems have matured a great deal, it has also become clear that the technology is no silver bullet: in the foreseeable future, the human element is going to remain an essential component in the analysis process, largely because our ability to evaluate the relevance of events *in context* of other activity is far superior to the one implemented in present-day technology. This evaluation is rendered more difficult by the fact that the technology currently does an insufficient job at distilling the amount of reported activity into a form whose volume is still comprehensible to humans and at the same time provides all relevant information necessary to understand the reported event in the full context of its occurrence.

We believe that much work remains to be done in helping the network analyst in that task. In this paper we present the *Brooery*,[1] a system to support the analysis of events reported by the Bro IDS [1]. We base our system on Bro because from the outset, Bro has taken a more differentiated approach to the detection problem than other IDSs, by separating *policy* (i.e., what events to report) from *mechanism* (i.e., how to extract the basic building blocks of events from the network). This separation turns out to be crucial in the analysis process, because the difference between relevant events and noise is often entirely defined by a site's policy. By deploying a monitoring policy in line with our understanding of relevance, we can reduce the volume of reported events from the outset. The Brooery presents events to the analyst through a graphical user interface. It allows quick drill-down to relevant details by allowing the analyst to switch between different log types, by the use of contextual navigation techniques, and by employing semantically strong hierarchical filtering that does not require external skills (such as SQL proficiency) from the analyst.

We first recapture Bro's current features in Section 2 to give the reader an intuition of the system the Brooery interfaces with. We present our requirements for the system in Section 3 before describing in detail our resulting architecture along with implementation details in Section 4. We then exemplify the application of our system in Section 5 and review related work in Section 6. The current state of the system and avenues for future work are discussed in Section 7 before we summarize the paper in Section 8.

2 Bro: A Distributed Event-Based Intrusion Detection System

Bro's architecture has remained faithful to the philosophy developed in the original paper [1]. A significant recent improvement has been the introduction of a communications framework as the basis of a more powerful event model suitable for distributed event communication [2, 3]. Figure 1 illustrates Bro's architecture.

2.1 Separation of Mechanism from Policy

A core idea of Bro is to split event detection mechanisms from event processing policies. Event generation is performed by *analyzers* in Bro's core: these analyzers operate continuously and trigger events asynchronously when relevant activity is observed. Examples include the establishment of a new TCP connection, or the request for a URL in an HTTP request. Bro's core contains analyzers for a wide range of network protocols such as TCP, UDP, FTP, HTTP, ICMP, SMTP, RPC, and others. Care is taken to minimize CPU load: only analyzers responsible for triggering the events used at the policy layer are actually enabled. Bro also provides a bidirectional *signature engine* for typical misuse-based intrusion detection: it matches byte string signatures against traffic flows and triggers an event whenever a signature matches [4].

Once an event is triggered, the engine passes it to the *policy layer*. Each Bro peer runs a policy configuration in its policy layer. This policy embodies the site's security policy, expressed in scripts containing statements in the special-purpose Bro scripting language. The language is strongly typed, procedural in style, and provides a wide range of elementary data types to facilitate the analysis of activity on a network. The policy layer maintains a large variety of state information about the activity currently observed on the network. For each event type, one or more *event handlers* are triggered that process events, possibly triggering new ones. Event types are defined by a *name* and a set of typed parameters that characterize individual events.

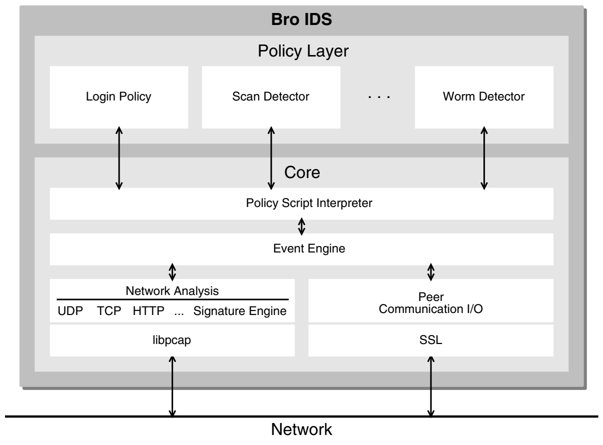


Figure 1: Architecture of the Bro IDS.

2.2 Event Logging

Event handlers may decide to *log* an event to persistent storage in a suitable machine- or human-readable format for later analysis. Bro's log management facility currently comprises two stages. First, at the policy level, Bro supports a basic notion of log files. These logs can be opened, closed, and printed to using `printf`-inspired functions. For notices and alarms, separate Bro policies take care of formatting the events appropriately before writing them out. The format of these entries is human-readable, but sufficiently structured for easy parsing. Second, log rotation is configured within the policy layer as well. This takes care of archiving the logs: rotation intervals, monitoring log file size and duration, labelling with start- and end timestamps, and file compression are all taken care of from within the policy layer.

2.3 Communication Framework & State Management

Bro's communication framework supports the serialization and transmission of arbitrary kinds of state between Bro instances. The driving idea behind its design is to allow the realization of *independent state* [2]: we should no longer think of state accumulated at the policy layer as a local concept, but rather as information dispersed and stored throughout the network. The communication model imposes no hierarchical structure. Examples of exchangeable state include triggered events, state kept in policy data structures, and the policy definitions themselves. For the purpose of this paper it is sufficient to think of the entities exchanged between peers as events, though that ignores a large part of its flexibility.

To interface other applications with Bro, we have implemented a lightweight, highly portable library supporting Bro's communication protocol called *Broccoli*,[2] that allows nodes which are not instances of the Bro IDS to partake in its event communication. Broccoli nodes can request, send, and receive Bro events just like Bro itself, but cannot be configured using Bro's policy language. A Broccoli node's policy has to be implemented in the client's code or through mechanisms such as configuration files.

3 The Brooery's Requirements

3.1 Usability Requirements

We have identified the following set of requirements for our system:

- Interoperability with Bro: The system should not require significant changes to Bro itself. Existing communication mechanisms should be leveraged as much as possible.
- Focus on investigation: The primary goal of the system is to enable the analysis of log archive content using a graphical interface, not to provide a real-time alert notification system. A highly interactive user interface, while clearly desirable, is thus not a primary requirement.
- Experimental prototyping: Support for rapid prototyping and experimentation with visualization techniques is more important at this stage than performance optimizations and long-term maintainability.
- Flexible filtering: The predominant problem in the analysis of network activity is the total volume of information. For this reason, effective filtering is essential. The analysis of Bro's log files has so far mostly happened at the shell prompt, and the effectiveness was essentially defined by the analyst's command of the typical text processing toolset: grep, awk, sed, and Perl, just to name few. Skilled use of these tools, while often unintuitive to other analysts, can be quite effective. The system should therefore aim at supporting this mindset in its filter management.
- Contextuality: The richness and diversity of events in Bro requires great flexibility from an analysis environment. The visual navigation should naturally guide the user at all times depending on the context of the currently inspected events, and provide mechanisms for quick drill-down to allow the analyst to focus on the relevant activity.
- Easy accessibility: At present, Bro is in day-to-day use throughout several large organizations around the planet. Such deployments require analyst access from multiple locations and using different platforms. We therefore prefer a standardized and widely available rendering mechanism that requires as little preconfiguration on the analyst's machine as possible.
- Spatial source independence: We would like to be able to select individual Bro nodes as data sources because we do not want to require that all data logging happen at a single place in the network.
- Representational source independence: Bro has traditionally created a set of text-based log files in order to record events for long-term storage. Two other forms of data storage are standard database back-ends and live state contained in running Bro nodes. We would like the system to support access to these uniformly.
- Different user sophistication levels: While Bro's design allows for much flexibility in its configuration, this freedom also means that its users need to spend more time to familiarize themselves with the system before they can use it efficiently and effectively. The user interface should support users of a wide range of sophistication levels, ranging from the occasional log inspection to operators who are intimately familiar with Bro policy development and day-to-day Bro maintenance.

3.2 Threat Model

The Brooery's main purpose is to present highly security-relevant information to the analyst, whose conclusions may have severe consequences for the operation of the network. We therefore need to be

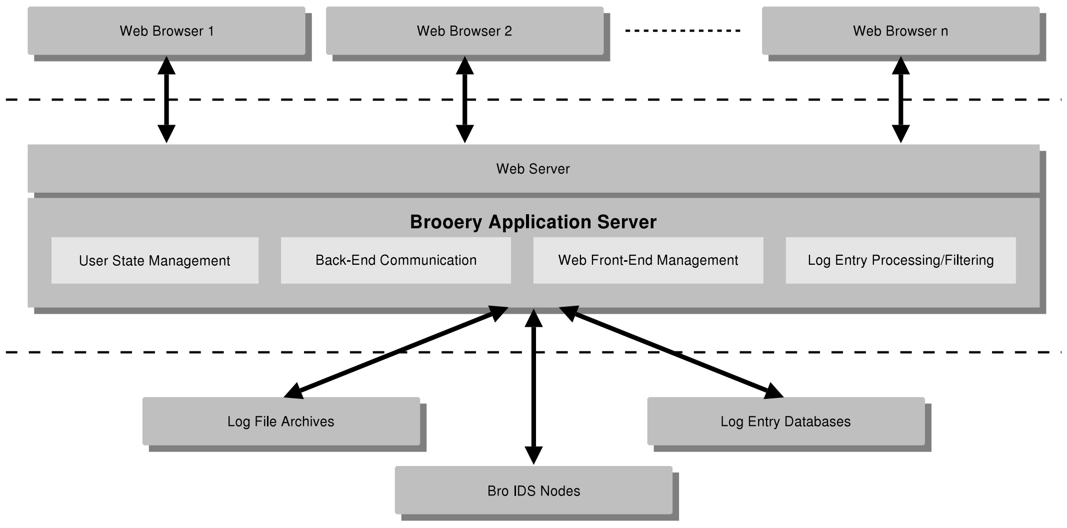


Figure 2: The Brooery's three-tiered Architecture.

aware of avenues attackers can follow in order to fool the analyst with false positives and negatives: an attacker might try to hide successful break-ins from the system, or equally dangerously, have the system report that a break-in did succeed where in fact there was none. The former causes the analyst to miss crucial information, while the latter would constitute a denial of service attack if for example machines had to be taken off line for investigation and recovery. The main attack surfaces are listed in the following and need to be protected carefully by the system:

- At the source. The log entry archives could be manipulated directly, introducing fake events or removing existing ones. Note that this is different from the typical case where IDSs are tricked into false positives or negatives; here, the storage system itself is subverted.

- In transit. Log entries need to be transferred and potentially filtered on their way from the archive to the analyst's console. An attacker with full control over the involved network flows could drop and introduce events at will, if the flows are not protected from tampering.

- During filtering & rendering. The system needs to process and filter log entries for presentation to the analyst. Similar to transit, an attacker who can modify the way the system itself operates can drop and introduce information or just cause the system in general to fail.

- At the destination. Since the application is mainly intended to read, process, and visualize existing information, the main benefit an attacker would gain from having access to the analyst's console is insight into what activity Bro nodes have been monitoring. When the system also permits the analyst to take administrative measures by updating running Bro nodes from the console, attackers with sufficient privileges to assume the role of an administrator could disable or attempt to crash individual Bro nodes.

4 The Brooery's Architecture

Given the requirements just outlined, we decided to implement our system in the three-tiered architecture illustrated in Figure 2. Analysts access the system through web browsers. The web server's back-end implements the core of the application, taking care of user interface rendering, user management, data source communication, and most importantly, the actual log entry processing. We will now discuss each of these components in more detail.

4.1 Web-based user interface

By using a web-based interface, we do away with the need to deploy stand-alone client applications, while at the same time avoiding any porting overhead that such an application might entail. Web-based interfaces fall short of the interactivity of full-blown client-side applications unless they employ heavy-weight Java Applets or typically unrobust JavaScripts. We want to avoid the use of such features to keep the list of requirements on the client side as small as possible. However, as outlined in the our requirements, the system is primarily meant as an analysis tool for *investigation* of past activity and not per se as a real-time alert *notification* tool. Furthermore, the web-based interface has the obvious advantage that external web-based services can be leveraged immediately through HTML linkage, for example to provide vulnerability information,[3] common TCP/UDP port usage,[4] or reports of scanning activity.[5] We have implemented the web front-end using the Open Source web site development framework Mason[6] and the Apache web server. This allowed us to (*i*) stay within the same language in which we have already accumulated a considerable amount of log analysis code and experience (see Section 4.4 below), and (*ii*) employ more advanced language features and mechanisms for structuring the components of the generated web pages than provided by other popular web site development solutions like for example PHP.[7]

4.2 Multiple communication back-ends

The Brooery is designed to support multiple communication back-ends for communicating with log archives in the form of text file repositories, databases, or live Bro agents. Log archives can reside on remote machines or locally. While log entry archives are clearly only useful for mining past activity, the third mechanism is useful for more general purposes. For example, it could be used to request resource usage summaries from running Bro instances (suitable policies are already part of the Bro distribution), or to adjust their current policies dynamically. All of this would happen within Bro's existing communications framework, making this approach potentially very powerful.

4.3 Management of user sessions

We support access to the system by multiple users at all times. For each user, the system stores his or her current analysis context, including log entries, filtering combinations, inspection time frames, and general user preferences. We currently maintain user identities in the form of user name and perform authentication using passphrases. User identities are used to present returning users with the environment they left earlier.

4.4 Log entry processing & filtering

This component is the core of the system and provides the domain knowledge necessary to manipulate Bro events and log entries. It is implemented in Perl, for four main reasons: first, a large body of well-maintained Perl modules has already been developed in the general context of the Bro project, so we can instantly leverage these efforts. Second, Perl is well suited for rapid prototyping, which we feel is very much the correct development philosophy at this point in time. Third, the CPAN Perl archive offers a vast set of modules for any kind of extension we are likely to need in the future. Fourth, it is easy to create bindings from Perl to native C, should the need arise. This could happen for purposes such as performance optimization, or integration of other components.

The API for log entry retrieval hides the details of the underlying mechanism. For the log processing engine, a log archive is structured into different *log types* representing the different logging domains and abstraction levels at which Bro reports events (e.g., connection summaries, notices, signature matches, or alarms). Each log type's archive is comprised of a set of *log slices*, each labelled with a start- and end timestamp identifying the timeframe it covers. Within a log slice, log entries can be obtained subject to a filtering condition (see Section 4.4.3). Each back-end implementation maps these abstractions to the actual API available for accessing a particular log archive. For accessing text file repositories, we have developed a simple log entry server that the corresponding Brooery back-end communicates with. This resembles in many ways a poor man's database implementation; we stress again that our focus at this point is not on obtaining optimal performance but getting a good feeling for the problem setting.

4.4.1 Timeframe & Log Type Selection and Log Navigation

Our current interaction model requires the user to start the investigation by selecting a timeframe of

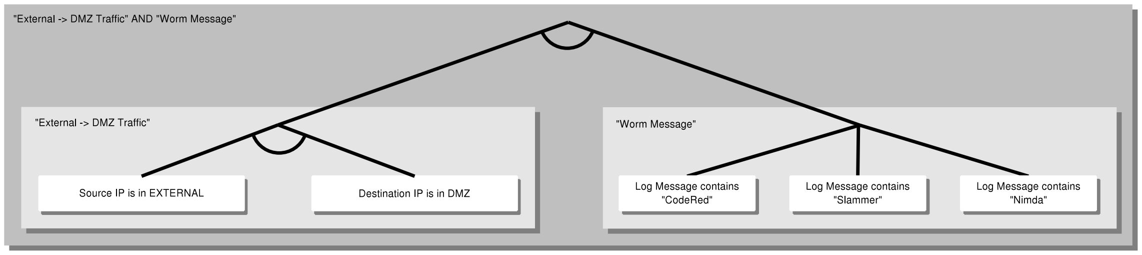

"External -> DMZ Traffic" AND "Worm Message"
"External -> DMZ Traffic"
Source IP is in EXTERNAL
Destination IP is in DMZ
"Worm Message"
Log Message contains "CodeRed"
Log Message contains "Slammer"
Log Message contains "Nimda"

Figure 3: Recursive filter combination using AND/OR-trees and types log entry components: the main filter matches if both the left and right subfilters match, the left subfilter matches if the source IP of an entry belongs to external networks and the destination IP is inside the DMZ, and the right subfilter matches when the message string of a log entry contains "CodeRed", "Slammer", or "Nimda".

activity on which to focus investigation. This timeframe defines a lower and upper bound on temporal relevance of log entries, across different log entry types. After selecting a particular log type, the system then obtains a list of log slices that contain log entries during the configured timeframe. From these log slices, up to a given maximum number of entries are then requested starting from a given timestamp. Within the configured timeframe, the user can then step forward and backward through the log entries, manipulating a configurable maximum number of log entries at any one time.

4.4.2 A Type System for Log Entry Components

In our log entry model, every component of a log entry has a *type*, inspired by the types provided by the Bro scripting language. The Brooery's type structure is richer than Bro's and more hierarchical: at the very least, every log entry component is of the root type "data" which provides textual operators such as "contains" and "does not contain". A large number of different types derive from this root, for example timestamps, flow sizes, port numbers, IP addresses, and protocol names. Further specializations exist for example for source and destination IP addresses.

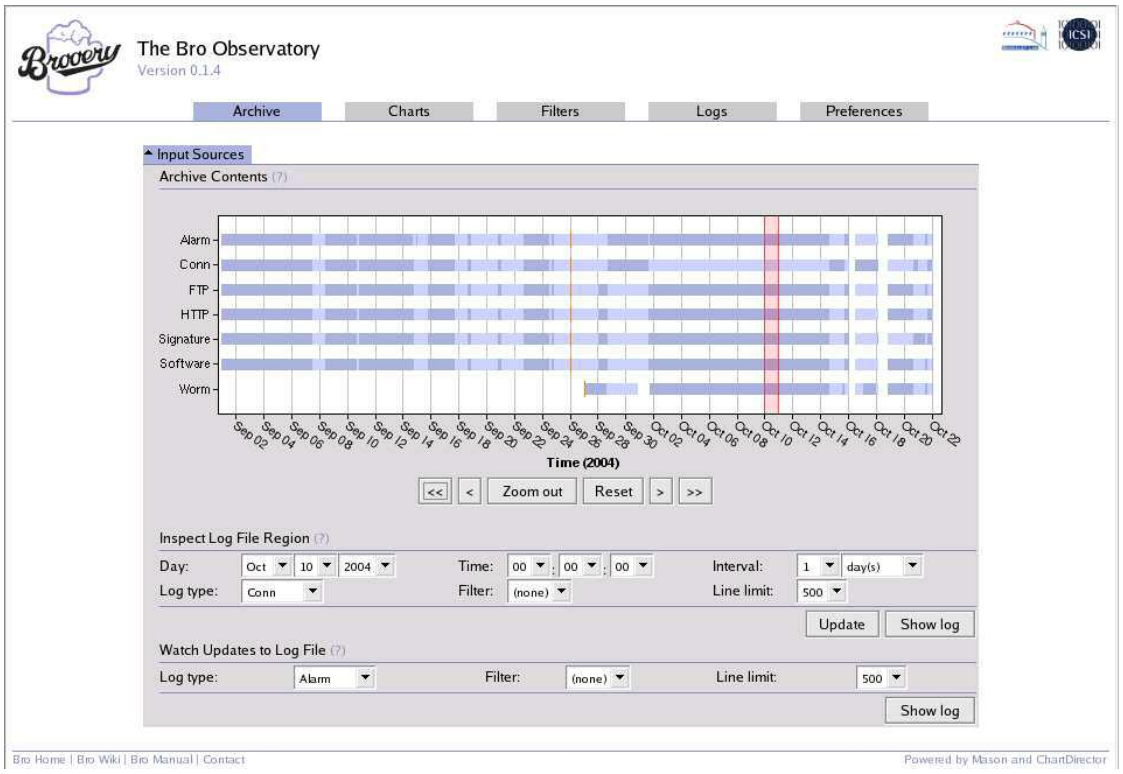
The benefits we gain from adhering to such a type model throughout all of our log types are *associativity*, *extensibility*, and *semantic processing*: regardless of the type of log we are currently investigating, a source IP address in one log file type will semantically represent the same as a source IP address in a different log type. This allows easy integration of future log types because only novel component types need to be integrated in the type hierarchy. The only information required to support a new log type is the sequence of the components' types. Furthermore, knowing that a log entry component represents for example a timestamp allows us to perform according operations on the component, in this case for example operations such as "earlier-than", "after", or "between".

4.4.3 Recursively Reusable AND/OR-Trees for Log Entry Filtering

The Brooery supports an elaborate concept of log entry matching based on AND/OR-trees, known from other applications such as attack trees [5]. The Brooery combines the expressiveness of conditions using AND/OR-trees with the strengths of the typed log entry components. For example, timestamps can be matched depending on whether they represent time earlier or later than a given timestamp, and IP addresses can be tested for (not) matching an address prefix.

A filter always consists of one ore more *filter parts*, each of which can either contain a filtering criterion as just described, or refer to another existing filter. The filtering results of all filter parts are then combined using Boolean conjunctions or disjunctions and lazy evaluation. Cyclic dependency detection prevents the user from configuring self-referring constructs. This approach to filter management allows the creation of arbitrarily nested filtering *hierarchies*, while ensuring easy re-use of existing filters. Figure 3 illustrates the concepts.

Note that a filter so far only represents a focusing mechanism; "filtering" is not meant to imply



Day: Oct 10 2004 Time: 00 : 00 : 00 Interval: 1 day(s)
Log type: Conn Filter: (none) Line limit: 500

Log type: Alarm Filter: (none) Line limit: 500

Figure 4: View of a Bro node's log archive.

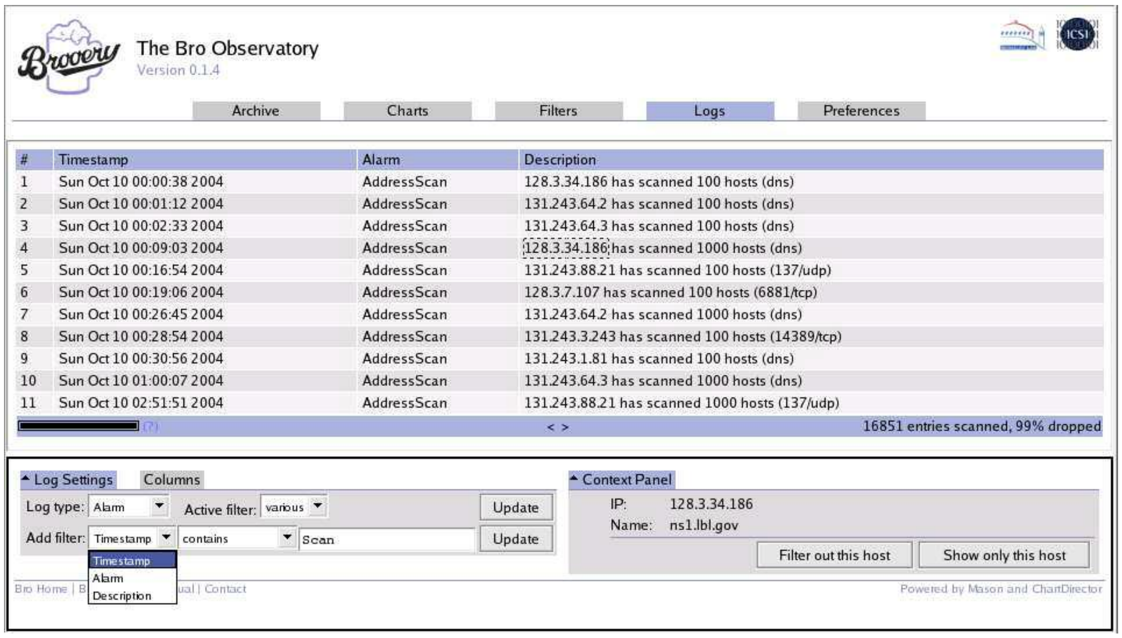
"dropping" at this point. Dropping a log entry due to successful filter evaluation is merely one of a range of conceivable *filter actions*; other examples include keeping a log entry, labelling it, and *aggregating* log entries by compressing all entries that are matched by a filter into a single abstract entry. We can summarize the aggregation by reporting in an abstract entry the number of entries it represents, while maintaining the common parts of those log entries visible.

The fact that multiple filters can trigger actions of different semantic meaning does imply that multiple filters may need to be active at the same time. For example, one filter could throw out unwanted entries, another one could aggregate the remaining ones. Integrating the output of multiple filters brings the potential of *conflict*: for example, one filter could declare that an entry is to be dropped while a second one asks for it to be kept. The way we solve this problem is through ordering: while the user may have multiple filters active at the same time, those filters have to be put in a sequence, and the first decision made in a conflict domain is decisive.

To allow the user to quickly weed out unwanted information and focus on the interesting entries, the Brooery also supports *incremental* filtering, i.e., the addition of new filter parts to a selected filter. At no time does the nature of the log entry storage shine through; for example, the user never has to resort to entering raw database queries.

4.5 Security Considerations

As outlined by our threat model, care must be taken to restrict the user base of the system to the intended individuals while preventing others from eavesdropping on or even tampering with the information flows. First of all, we assume that when an attacker manages to break into one of the hosts running Bro, or one of the machines storing a log archive, it is unlikely that we can prevent a determined intruder from causing serious damage to the system. Therefore, the security precautions of Bro nodes and log archives remain unchanged, regardless of whether these systems are interfaced with the Brooery or not.



#	Timestamp	Alarm	Description
1	Sun Oct 10 00:00:38 2004	AddressScan	128.3.34.186 has scanned 100 hosts (dns)
2	Sun Oct 10 00:01:12 2004	AddressScan	131.243.64.2 has scanned 100 hosts (dns)
3	Sun Oct 10 00:02:33 2004	AddressScan	131.243.64.3 has scanned 100 hosts (dns)
4	Sun Oct 10 00:09:03 2004	AddressScan	128.3.34.186 has scanned 1000 hosts (dns)
5	Sun Oct 10 00:16:54 2004	AddressScan	131.243.88.21 has scanned 100 hosts (137/udp)
6	Sun Oct 10 00:19:06 2004	AddressScan	128.3.7.107 has scanned 100 hosts (6881/tcp)
7	Sun Oct 10 00:26:45 2004	AddressScan	131.243.64.2 has scanned 1000 hosts (dns)
8	Sun Oct 10 00:28:54 2004	AddressScan	131.243.3.243 has scanned 100 hosts (14389/tcp)
9	Sun Oct 10 00:30:56 2004	AddressScan	131.243.1.81 has scanned 100 hosts (dns)
10	Sun Oct 10 01:00:07 2004	AddressScan	131.243.64.3 has scanned 1000 hosts (dns)
11	Sun Oct 10 02:51:51 2004	AddressScan	131.243.88.21 has scanned 1000 hosts (137/udp)

< > 16851 entries scanned, 99% dropped

Figure 5: Alarm log entries, filtering the "Alarm" column for entries containing "Scan".

Regarding the communication between the various nodes interacting with the Brooery, we use two different levels of restrictiveness: inside the Bro network, we can assume that the communicating entities know each other's identities. Besides employing SSL encryption, we can therefore require mutual authentication of the communicating peers through certificates. The web browsers accessing the system are still required to use encrypted connections via HTTPS, but as mentioned in Section 4.3 we drop the requirement for client-side certificates and resort to weaker authentication in the form of user names & passphrases. Furthermore, we aim to restrict the reachability of the involved hosts to a minimum whenever possible, for example by only making the ports on which Bro data sources can be tapped available on a separate network.

5 Usage Example

We will now give a quick but illustrative example of user interaction with the Brooery. Let us a assume that we have been informed that on October 10, several users have noticed unusual connection attempts to their machines. Our goal now is to find out whether any relevant scanning activity was detected that day. Figure 4 shows a Bro node's log archive, and we can see that the node has plenty of information for October 10: there are 7 different log types ranging from alarms over connection summaries to worm events, reaching in time from the beginning of September to the present at the time the screenshot was taken. Horizontal blue bars indicate the timeframes during which the Bro node was logging events of a particular type. The blue color is differentiated into two different shades, with each color switch indicating the start of a new log slice. Note that the bars cover the timeframe when the IDS was *ready to log*, not the timeframe from the first logged event to the last one — an important semantical difference. For example, we can see that the system was not monitoring at all for several hours on October 16 and 18, and that worm event logging was introduced on September 27. The small vertical orange lines delineate compressed from uncompressed archival, i.e., log entries residing on the left side of an orange line are stored in compressed fashion.

We specify a time interval covering that day, and look at the contents of the alarms log. We then add a filter on the alarm names that matches all entries containing "Scan", and obtain the result shown in Figure 5. As can be seen, several hosts have scanned a large number of machines, and we can now continue our analysis by looking at the connection summaries for each of those hosts during the given time period. To do so, we click on any of the shown IP addresses, add a filter part for that address using the option the context panel for IP addresses provides for this purpose, and switch the log type to the connection logs.

6 Related Work

In [6], Hoagland and Staniford proposed SnortSnarf, a web-based console for analyzing Snort alerts [7]. ACID[8] is similar and additionally allows the generation of charts and statistics. Our system is superficially similar to these, however our system is more comprehensive in that it (*i*) can manage a wider variety log types and (*ii*) structures log entries more thoroughly due to typed column entries for stronger semantic filtering across different log types. Sguil[9] is close to our system in the sense that it acknowledges the need for providing contextual information for alerts such as connection logs and packet content. Our system differs from theirs in that filter management in Sguil is less intuitive for the user (who has to resort to SQL statements); also, Sguil is implemented in Tcl/Tk and therefore not as readily accessible as our web-based interface. A number of other user interfaces for Snort exist; they are typically geared towards support for signature management and do not provide the flexibility to deal with the wide range of log information provided by Bro. In the commercial space, user interfaces are often bundled with IDS products directly or offered in the general Security Incident Management domain. As our focus is on open-sourced solutions, we do not review the commercial domain thoroughly in the scope of this paper.

7 Discussion & Future Work

The Brooery is work in progress and a prototyping testbed. We use it for experimentation with different models for analyzing log information and therefore feel it is important to point out that we are currently primarily interested in different metaphors for manipulating the log information; aspects such as performance optimization remain secondary. So far we have found the graphical instruments realizable using HTML sufficient for our needs; it will be interesting to see if this observation will apply to future extensions to the system as well.

We currently see two main avenues for future work. First, we need to augment Bro with a database logging component that does not require fundamental modifications of Bro's logging component. Database-driven archival is very much a necessity for robust log entry storage & retrieval, and, of course, performance. Text-file based storage, while familiar and to a certain degree manageable at the command line, restricts performance and can sometimes pose technical difficulties. One avenue we are considering for achieving this is to turn the act of logging an event into an event itself. That way, the implementation of the logging mechanism would remain up to individual event handlers, could happen in multiple ways in parallel, and other event logging systems (including the Brooery) could tap into the stream of logged events using e.g. Broccoli and the existing event communications framework. This approach would thus fit very nicely into Bro's model. Depending on the implementation of the event handlers responsible for processing such logging events, the events would then be stored in a text file, a database, or processed in some other way. The Brooery's log entry model is geared towards easy mapping onto relational structures; the main difference her are the semantically stronger types used at within the log entry engine.

Second, we intend to investigate the requirements for effective analysis of distributed events. We are currently correlating events originating on multiple Bro nodes using Bro's event communication framework; however, it is not yet clear to us what will turn out to be the best visual metaphor for controlling this correlation and visualizing the results.

8 Summary

We have presented the Brooery, a three-tiered experimental prototyping platform for graphical analysis of network activity reported by instances of the Bro IDS. The system provides contextually relevant drill-down features and supports different Bro log archival back-ends; semantically strong and reusable log entry matching based on AND/OR trees; filtering, labelling, and aggregation of log entries; and hierarchically typed log entry components. The Brooery's development is fully open sourced under a BSD license. More details can be found at `http://www.icir.org/twiki/bin/view/Bro/BrooeryGUI`.

Acknowledgements

This work is carried out in collaboration with Intel Research Cambridge, ICIR, and Lawrence Berkeley National Laboratory. It was supported in part by the U.S. National Science Foundation grant STI-0334088, and the U.S. Department of Energy. We would like to thank Vern Paxson, Jon Crowcroft, and the other Bro developers for helpful discussion and feedback, and the occasional brainstorming in real-world "brooery" environments.

References

[1] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23-24):2435–2463, 1998.

[2] Robin Sommer and Vern Paxson. Exploiting Independent State For Network Intrusion Detection. Technical Report TUM-I0420, TU München, 2004.

[3] Christian Kreibich and Robin Sommer. Policy-controlled Event Management for Distributed Intrusion Detection. In *Proceedings of the 4th International Workshop on Distributed Event-Based Systems (DEBS'05)*, June 2005.

[4] Robin Sommer and Vern Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Proc. 10th ACM Conference on Computer and Communications Security*, 2003.

[5] *Secrets and Lies*, pages 318–333. John Wiley and Sons, New York, 2000.

[6] James A. Hoagland and Stuart Staniford. Viewing IDS alerts: Lessons from SnortSnarf. Technical report, Silicon Defense, Nov 2000.

[7] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Conference on Systems Administration*, pages 229–238, 1999.

NetState : A Network Version Tracking System[*]

Nancy Durgin Yuqing Mai Jamie Van Randwyk

Sandia National Laboratories
Livermore, CA 94550

nadurgi@sandia.gov yuqingm@hotmail.com jvanran@sandia.gov

Abstract

Network administrators and security analysts often do not know what network services are being run in every corner of their networks. If they do have a vague grasp of the services running on their networks, they often do not know what specific versions of those services are running. Actively scanning for services and versions does not always yield complete results, and patch and service management, therefore, suffer. We present NetState, a system for monitoring, storing, and reporting application and operating system version information for a network. NetState gives security and network administrators the ability to know what is running on their networks while allowing for user-managed machines and complex host configurations. Our architecture uses distributed modules to collect network information and a centralized server that stores and issues reports on that collected version information. We discuss some of the challenges to building and operating NetState as well as the legal issues surrounding the promiscuous capture of network data. We conclude that this tool can solve some key problems in network management and has a wide range of possibilities for future uses.

1 Introduction

As computer networks grow larger, it becomes more difficult to manage those networks. It is increasingly difficult for information technology (IT) departments to manage large numbers of computers and similar devices on their networks. As users become more savvy, it is more difficult to control the network services that users run on their computing devices. In addition, viruses, Trojan-horses and worms may install "back-door" network services. Firewall and corporate policies are only able to control the spread of network services to a limited degree.

Because IT departments cannot always control which network services are being run on their networks, they must find a way to identify which services are being run on which devices. In the past, port scanning (using a tool such as Fyodor's *Nmap* [1]) was a reasonably airtight technique used to identify services running on a given network-enabled device. Now users install common network services on non-standard ports to get around corporate firewall restrictions. Some users install multiple operating systems on a single computer, rendering port scans incomplete. Trojan-horses use proprietary network protocols on seemingly random ports to conduct their nefarious activity.

Not knowing what services are running on one's network makes patch management and service management extremely difficult. This can open network devices up to compromise because the IT staff cannot identify and patch all instances of the affected service after a new vulnerability announcement.

We built NetState to passively monitor, store, and report application and operating system version information for a network. NetState includes sniffer modules that monitor traffic across a network, a backend database for storing service name and version information, and a GUI client for querying the database. NetState was built for internal use but is now being made available in the public domain.

We have organized the rest of this paper as follows: In Section 2 we survey existing open source and commercial tools that attempt to catalog service versions in existence on a network. Then, in Section 3 we discuss both our design requirements and our implementation. Our discussion of the implementation describes the three main components of NetState: the NetState Sniffer, the NetState Server, and the data query interfaces. In Sections 4, 5, and 6 we describe the performance of NetState

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

in a low-bandwidth network and relate our experiences and perceived challenges in using NetState on a day-to-day basis. Next, in Section 7, we give an overview of legal issues surrounding the "sniffing" of data in both the employer-employee and Internet service provider-customer scenarios. We conclude by discussing future work in Section 8.

2 Related Work

When we surveyed the commercial and open source communities for software to perform our desired functions we did not find anything that fit our requirements.

Several companies sell products that monitor and record network traffic for further analysis [2, 3]. Presumably, these products offer the ability to report network service versions, but we did not test for this. Company literature was also not clear in identifying the existence of this feature. We did not fully evaluate these products because their overall utility was far more than we needed.

Novell sells the desktop management product *ZENworks* that allows centralized management of many independent systems via a *management agent* running on those machines [4]. From a central location a network or security administrator can enforce a standard desktop environment, migrate personal settings, deploy software patches, and monitor system performance. The central management server provides a network analyst with in-depth information about each of the managed hosts. This includes information about network service versions. We would not be able to rely on such software to accurately represent our entire network though. Our laboratory research environment demands that some systems be managed solely by the researcher that owns them, often meaning that remote management utilities cannot be installed by our desktop support team. More importantly, ZENworks only supports the NetWare, Windows, and Linux operating systems. Our networks are home to systems running many operating systems beyond those supported by ZENworks, also including Linux distributions not officially supported. Finally, the capabilities of ZENworks are far beyond what we wanted to implement; the Novell software would duplicate functionality already established by competing products on our networks.

Nmap recently introduced a network service version scanning feature. Using the '-sV' or '-A' options, an analyst can identify the application name and version information when available. Nmap performs this inquiry for each open port that it discovers during the port scanning procedure. The community involvement with keeping the service version database up-to-date is especially valuable to Nmap. Nmap is designed to be an active scanning tool though. It cannot detect when a multi-boot system has been booted into a different operating system or when a machine that is often powered off has been powered on. In these situations, Nmap could fail to identify open and potentially vulnerable network services.

We have decided to release the NetState source code to the open source community for several reasons. Primarily, we believe that secure public networks are of benefit to everybody. A tool that allows network administrators to be more aware of the behavior of the machines on their networks is one more step towards this goal. In addition, while we have implemented version detection for many common protocols, we feel the open source community can contribute support for additional protocols or improve upon the current detection methods.

3 Architecture

Our design was formed after discussing goals with our security analysts, studying our existing security architecture and evaluating a prototype tool that we had written. We first discuss some of our design requirements and then describe how we met those in our implementation.

3.1 Design Requirements

- **Fits into our existing network security framework:** The server computer should exist on a private security network. Only security personnel, other authorized personnel, and machines owned by security should be able to access this machine via the network. The network traffic being monitored should be on a separate network.

- **Comprehensive data collection:** Traffic should be monitored at every network ingress and egress point. This includes wide-area Internet links, VPN concentrators, modem pools, wireless access points, point-to-point links with collaborators, etc.

- **Pertinent information only:** All information about network application versions should be collected for all computers, but we should not store any more than that. We should be able to determine which application version is/was running on which port on each device at any point in time.

- **Major network applications:** We should track information for "major" applications on our network. This includes services and client applications that are commonly used or are mission-critical. We implement this in such a way as to allow expansion to additional applications in the future.

- **Passive monitoring:** All version information should be collected passively by monitoring all the traffic on the target network.

3.2 Passive vs. Active Scanning

A major distinction of NetState's design is that it uses passive scanning techniques as opposed to the active techniques employed by tools such as *Nmap* and *Nessus* [5]. While active scanning techniques can often yield more detailed or precise data, for example by sending specially crafted packets that yield a definitive signature, passive scanning offers several advantages:

- Active scanning tools are "noisy", creating additional, and often unnecessary, traffic on a network.
- If a particular machine is turned off, a single machine boots multiple operating systems, or multiple machines share the same IP addresses at different times (via DHCP), it is difficult to guarantee detecting these situations via active scans. Since passive scanning monitors network traffic at *all* times, it yields information about what actually happens on the network, rather than just a snapshot of what the network looked like when the scan was performed.

These active scanning tools can still be used to assist NetState in gathering application version information. We can populate the NetState database by performing an active scan that receives its results under the nose of a NetState Sniffer.

3.3 Implementation

Our design goals led us to implement a distributed system consisting of several modular programs all working together as NetState. The architecture is shown in Figure 1.

The core of our system is a server process that accepts network traffic information from distributed Sniffer processes and places the information into a database. The NetState Server receives connections from NetState Sniffers via a private "security" network. The Server receives application version information over those connections and stores the information in a database. These connections could be established over the open network as well, but NetState needs built-in authentication before that is practical. The Server also responds to queries from authorized clients that are allowed to access the application version database. Access control is maintained using operating system-level firewall rules.

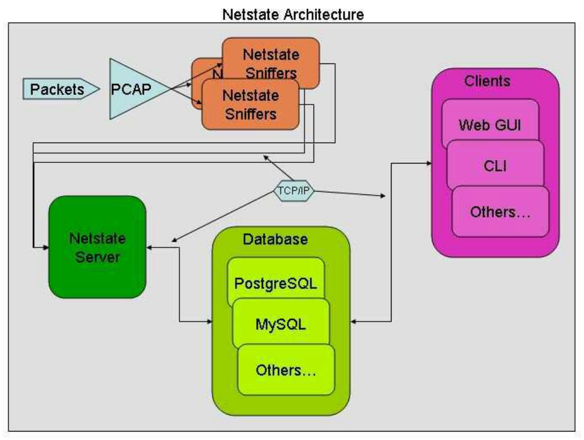


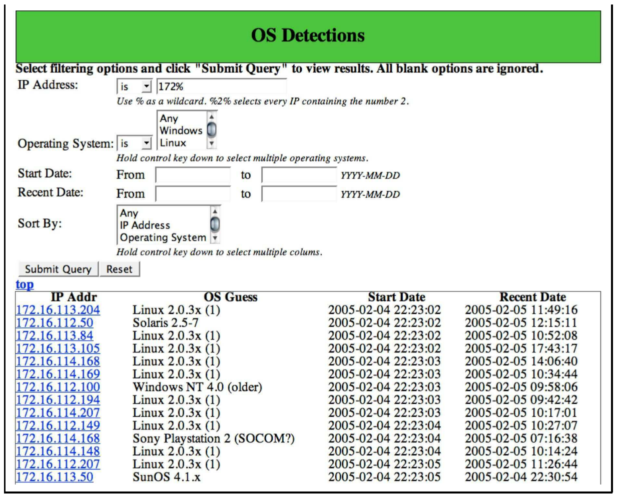
Figure 1: The NetState architecture includes distributed *Sniffers* that capture Ethernet frames and parse them for version information. The Sniffers send this information on to the NetState *Server* which inserts the data into a database. The NetState Web GUI (graphical user interface) and the database CLI (command-line interface) can be used to query the database and retrieve version information.

3.4 NetState Sniffers

The NetState Sniffers are designed to be deployed in many locations on a network. The Sniffers capture network traffic using libpcap [6], the packet capture library available for most UNIX and UNIX-like operating systems. The Sniffers listen passively on a network interface that is given access to all traffic on the to-be-monitored network link, whether by a switch's port-mirroring function, a network tap or some other method.

Operating system detection is performed using the open source program *p0f* (version 2) [7] [Figure 2]. (The data used for our Web GUI figures were PCAP files collected by MIT Lincoln Labs as part of the DARPA Intrusion Detection Evaluation project [8]. We replayed the traffic on a private network using Tcpreplay [9].) We modified p0f to tightly integrate it into NetState, calling it directly as a subroutine. OS detection is performed at the beginning of each new connection, on the synchronize (SYN) packet.

Application version detection is performed by looking at the first few data packets of a connection [Figures 3, 4]. The first data packet is examined for "magic strings" which indicate it likely contains traffic of a specific type. For example the magic string for the FTP and NNTP protocols is the number "202" at the beginning of a line, while the magic string for an HTTP server is "HTTP" at the beginning of a line. If a magic string is found, then further processing is done to find a version string in that packet or from one of the next few packets, depending


OS Detections

Select filtering options and click "Submit Query" to view results. All blank options are ignored.

IP Address: is 172%
Use % as a wildcard. %2% selects every IP containing the number 2.

Operating System: is Any / Windows / Linux
Hold control key down to select multiple operating systems.

Start Date: From ____ to ____ *YYYY-MM-DD*
Recent Date: From ____ to ____ *YYYY-MM-DD*

Sort By: Any / IP Address / Operating System
Hold control key down to select multiple colums.

Submit Query | Reset

top

IP Addr	OS Guess	Start Date	Recent Date
172.16.113.204	Linux 2.0.3x (1)	2005-02-04 22:23:02	2005-02-05 11:49:16
172.16.112.50	Solaris 2.5-7	2005-02-04 22:23:02	2005-02-05 12:15:11
172.16.113.84	Linux 2.0.3x (1)	2005-02-04 22:23:02	2005-02-05 10:52:08
172.16.113.105	Linux 2.0.3x (1)	2005-02-04 22:23:02	2005-02-05 17:43:17
172.16.114.168	Linux 2.0.3x (1)	2005-02-04 22:23:03	2005-02-05 14:06:40
172.16.114.169	Linux 2.0.3x (1)	2005-02-04 22:23:03	2005-02-05 10:34:44
172.16.112.100	Windows NT 4.0 (older)	2005-02-04 22:23:03	2005-02-05 09:58:06
172.16.112.194	Linux 2.0.3x (1)	2005-02-04 22:23:03	2005-02-05 09:42:42
172.16.114.207	Linux 2.0.3x (1)	2005-02-04 22:23:03	2005-02-05 10:17:01
172.16.112.149	Linux 2.0.3x (1)	2005-02-04 22:23:04	2005-02-05 10:27:07
172.16.114.168	Sony Playstation 2 (SOCOM?)	2005-02-04 22:23:04	2005-02-05 07:16:38
172.16.114.148	Linux 2.0.3x (1)	2005-02-04 22:23:04	2005-02-05 10:14:24
172.16.112.207	Linux 2.0.3x (1)	2005-02-04 22:23:05	2005-02-05 11:26:44
172.16.113.50	SunOS 4.1.x	2005-02-04 22:23:05	2005-02-04 22:30:54

Figure 2: These query results, using the NetState Web GUI, show a partial listing of the operating system versions detected on the 172.*.*.* network. The dates indicate when the operating system was first detected and when it was most recently detected.

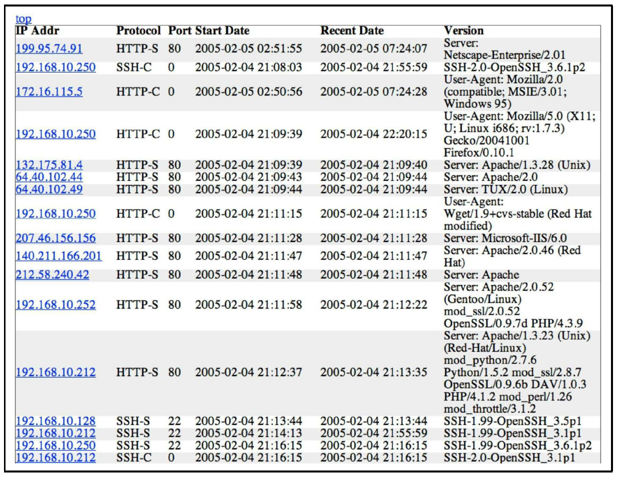
on the specific protocol.

In most cases the version string that is stored in the database for applications is simply the entire string in which the version appears. No attempt is made to pull a numeric value out of a string, because in most cases the format of the version string is not well-defined, but instead, tends to follow common conventions.

In a few cases, e.g. for the file transfer protocol (FTP) and the simple mail transport protocol (SMTP), some implementations append a time/date stamp to the version. Since the timestamp would cause each version string for sessions occurring at different times with the same server to be logged as a new version, this information is stripped off. These sorts of issues need to be discovered and handled on a case-by-case basis.

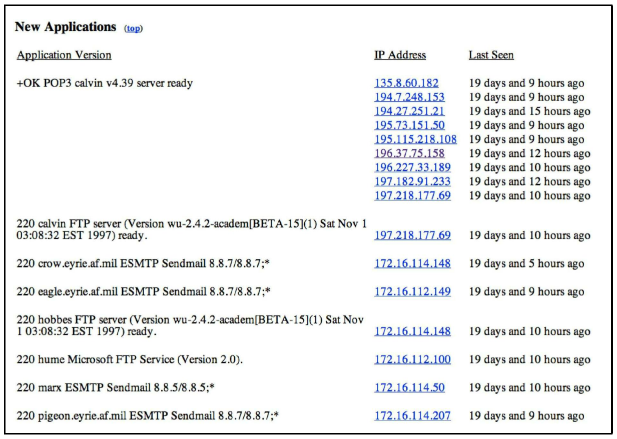
Currently NetState does not use the port number to infer that a particular application is running. It will find hypertext transfer protocol (HTTP) traffic on any port, FTP on any port, etc. By not using the port number as a "hint", we are more restricted in what applications we can currently detect, but since we want to be able to detect rogue applications on unusual ports, this seemed like the correct design decision.

The NetState Sniffer keeps a cache of recently seen version numbers by IP address and port. If a version string is detected that was seen recently, the timestamp is updated internally in the Sniffer but not updated to the Server component right away. A timeout can be configured to control how often the cache updates. This caching feature was added to improve database performance on busy networks by reducing the number of


top

IP Addr	Protocol	Port	Start Date	Recent Date	Version
199.95.74.91	HTTP-S	80	2005-02-05 02:51:55	2005-02-05 07:24:07	Server: Netscape-Enterprise/2.01
192.168.10.250	SSH-C	0	2005-02-04 21:08:03	2005-02-04 21:55:59	SSH-2.0-OpenSSH_3.6.1p2
172.16.115.5	HTTP-C	0	2005-02-05 02:50:56	2005-02-05 07:24:28	User-Agent: Mozilla/2.0 (compatible; MSIE/3.01; Windows 95)
192.168.10.250	HTTP-C	0	2005-02-04 21:09:39	2005-02-04 22:20:15	User-Agent: Mozilla/5.0 (X11; U; Linux i686; rv:1.7.3) Gecko/20041001 Firefox/0.10.1
132.175.81.4	HTTP-S	80	2005-02-04 21:09:39	2005-02-04 21:09:40	Server: Apache/1.3.28 (Unix)
64.40.102.44	HTTP-S	80	2005-02-04 21:09:43	2005-02-04 21:09:44	Server: Apache/2.0
64.40.102.49	HTTP-S	80	2005-02-04 21:09:44	2005-02-04 21:09:44	Server: TUX/2.0 (Linux)
192.168.10.250	HTTP-C	0	2005-02-04 21:11:15	2005-02-04 21:11:15	User-Agent: Wget/1.9+cvs-stable (Red Hat modified)
207.46.156.156	HTTP-S	80	2005-02-04 21:11:28	2005-02-04 21:11:28	Server: Microsoft-IIS/6.0
140.211.166.201	HTTP-S	80	2005-02-04 21:11:47	2005-02-04 21:11:47	Server: Apache/2.0.46 (Red Hat)
212.58.240.42	HTTP-S	80	2005-02-04 21:11:48	2005-02-04 21:11:48	Server: Apache
192.168.10.252	HTTP-S	80	2005-02-04 21:11:58	2005-02-04 21:12:22	Server: Apache/2.0.52 (Gentoo/Linux) mod_ssl/2.0.52 OpenSSL/0.9.7d PHP/4.3.9
192.168.10.212	HTTP-S	80	2005-02-04 21:12:37	2005-02-04 21:13:35	Server: Apache/1.3.23 (Unix) (Red-Hat/Linux) mod_python/2.7.6 Python/1.5.2 mod_ssl/2.8.7 OpenSSL/0.9.6b DAV/1.0.3 PHP/4.1.2 mod_perl/1.26 mod_throttle/3.1.2
192.168.10.128	SSH-S	22	2005-02-04 21:13:44	2005-02-04 21:13:44	SSH-1.99-OpenSSH_3.5p1
192.168.10.212	SSH-S	22	2005-02-04 21:14:13	2005-02-04 21:55:59	SSH-1.99-OpenSSH_3.1p1
192.168.10.250	SSH-S	22	2005-02-04 21:16:15	2005-02-04 21:16:15	SSH-1.99-OpenSSH_3.6.1p2
192.168.10.212	SSH-C	0	2005-02-04 21:16:15	2005-02-04 21:16:15	SSH-2.0-OpenSSH_3.1p1

Figure 3: These query results show the detected network services and their corresponding versions and port numbers on the local network. The "start date" and "recent date" columns indicate when the particular service was first detected and most recently detected, respectively. NetState does not track port numbers for client version strings, so the ports for the HTTP and SSH clients are filled with a placeholder "0".


New Applications (top)

Application Version	IP Address	Last Seen
+OK POP3 calvin v4.39 server ready	135.8.60.182	19 days and 9 hours ago
	194.7.248.153	19 days and 9 hours ago
	194.27.251.21	19 days and 15 hours ago
	195.73.151.50	19 days and 9 hours ago
	195.115.218.108	19 days and 9 hours ago
	196.37.75.158	19 days and 12 hours ago
	196.227.33.189	19 days and 10 hours ago
	197.182.91.233	19 days and 12 hours ago
	197.218.177.69	19 days and 10 hours ago
220 calvin FTP server (Version wu-2.4.2-academ[BETA-15](1) Sat Nov 1 03:08:32 EST 1997) ready.	197.218.177.69	19 days and 10 hours ago
220 crow.eyrie.af.mil ESMTP Sendmail 8.8.7/8.8.7;*	172.16.114.148	19 days and 5 hours ago
220 eagle.eyrie.af.mil ESMTP Sendmail 8.8.7/8.8.7;*	172.16.112.149	19 days and 9 hours ago
220 hobbes FTP server (Version wu-2.4.2-academ[BETA-15](1) Sat Nov 1 03:08:32 EST 1997) ready.	172.16.114.148	19 days and 10 hours ago
220 hume Microsoft FTP Service (Version 2.0).	172.16.112.100	19 days and 10 hours ago
220 marx ESMTP Sendmail 8.8.5/8.8.5;*	172.16.114.50	19 days and 10 hours ago
220 pigeon.eyrie.af.mil ESMTP Sendmail 8.8.7/8.8.7;*	172.16.114.207	19 days and 9 hours ago

Figure 4: These query results show the IP addresses detected to be running a given network service and version number (listed by services). Also included is how long ago that service was last detected. NetState does not currently remove host names from version strings; thus, identical versions of Sendmail appear to be different versions to NetState.

database updates performed by the server component. The result of the caching is that the most-recently-seen time value in the SQL database may not be completely up-to-date at any given time.

The NetState Sniffer maintains information on all active connections, as well as the version cache information, in memory. Its memory fingerprint can be quite large, approaching 512 MB on a busy network (e.g. 1000+ hosts). It loads some configuration files (e.g. the p0f fingerprints) from disk but does not maintain any state on disk.

3.5 NetState Server

The NetState Sniffers capture data off of the network including the application version string, IP address and port number. This three-tuple of information is then sent to the NetState Server. The Server collects this information and writes it to a database along with the current date and time. If the three-tuple creates a new application-version entry, the timestamp is also stored both in a "first-seen" field and a "most-recently-seen" field. If the three-tuple already exists in the database, the Server updates the "most-recently-seen" field with the current timestamp. The database thus stores five-tuple entries containing the application version string, IP address, port number, first-seen timestamp and last-seen timestamp.

The NetState Server is implemented as a daemon listening on a socket on a specific TCP port (the default is 2003) for messages from a Sniffer. If it detects a new connection on the port, it spawns a copy of itself to handle that connection. The Server is implemented as a simple loop that translates messages received from the Sniffer into appropriate SQL database commands to update the database. It does not have any significant memory or disk structures to maintain (other than the SQL database itself).

The database may be located on the same system as the NetState Server, or it may be located on a separate back-end database server. NetState currently supports both the MySQL and PostgreSQL open-source databases. Each of the NetState components was designed to be run on Linux and BSD-derived operating systems. Tested operating systems include *RedHat Linux 9.0*, *Fedora Core 2* and *FreeBSD 4.8*.

3.6 Reporting Interfaces

A Web interface can be used to query the NetState Server for information regarding the service applications on a network. The client includes functionality for several "canned" queries that answer questions including:

- What versions of software (for supported protocols) are currently running on IP x?
- What ports are open on IP x?
- What are all the versions of protocol x (e.g. SSHD) running on network y?
- What IPs on network x are running protocol/version y less than version z (e.g. *OpenSSH* versions < protocol 2)?

In addition to the Web interface, scripts can be written in any language with SQL library support (e.g. *Perl*), to generate reports about the hosts on the network in any desired format.

Some examples of typical SQL queries are shown in Tables 1 and 2. These are the types of queries that can be integrated into a graphical GUI or a Perl script, as desired. The query in Table 1 lists the OS strings from all the machines in the database. The record name is `os_detect`, and the string for the `os_version` field comes from the p0f fingerprint file.

Another example of a useful query is shown in Table 2, which shows the software version for all the machines on the network that are running an HTTP server. Note that in this example, there are several duplicate entries for a particular IP address. This can happen when a web-proxy is being used. In this query, `version` is the name of the database record. The `version` field is the string that was detected by NetState. The `description` field is a mnemonic human-readable field that is determined by NetState. Note that "HTTP-S" refers to "HTTP Server"; we use HTTP-C to refer to the version for the client side. It does *not* mean "secure HTTP" (the *HTTPS* protocol).

Another interesting query is

```
select ip_addr_dot, port, description,
version from version where description =
'HTTP-S' where port != 80;
```

This query will list all the HTTP servers on the network that are not running on the standard port 80.

4 Performance

The first version of NetState did not do any internal caching of version information in the Sniffer component. The information for each version string was handed directly to the NetState Server, where duplicate version strings were handled by updating the "most recent time seen" field in the database record. Performance testing indicated that on a busy network the SQL queries would bottleneck the system. A version cache was added to the Sniffer component to mitigate this bottleneck. The cache works by watching for a version string associated with an IP address that is identical to one that was seen

```
mysql> select ip_addr_dot, recent_date, os_version from os_detect;
+-----------------+---------------------+---------------------------------------------------------+
| ip_addr_dot     | recent_date         | os_version                                              |
+-----------------+---------------------+--------------------------------------------------------+|
| 192.168.10.182  | 2003-11-26 15:18:44 | Linux 2.4.2 - 2.4.20                                    |
| 192.168.10.202  | 2003-11-20 17:20:33 | Linux 2.4.2 - 2.4.20                                    |
| 192.168.10.97   | 2003-11-25 14:12:36 | Windows XP Pro, Windows 2000 Pro                        |
| 192.168.10.20   | 2003-11-25 09:33:49 | Windows 98 or Windows 2000 SP4                          |
| 192.168.10.31   | 2003-11-21 15:33:42 | FreeBSD 5.0-RELEASE or Macintosh PPC Mac OS X (10.2. ...|
| 192.168.10.31   | 2003-11-21 15:33:45 | Macintosh PPC Mac OS X (10.2.1 and v?)                  |
| 192.168.10.5    | 2003-11-21 16:55:51 | Windows 2000                                            |
| 192.168.10.4    | 2003-11-25 16:55:16 | Windows 98 or Windows 2000 SP4                          |
| 192.168.10.218  | 2003-11-26 12:00:40 | Windows 2000                                            |
 (...)
| 192.168.10.51   | 2003-11-26 11:24:59 | Windows 98 or Windows 2000 SP4                          |
| 192.168.10.84   | 2003-11-25 16:04:28 | Windows XP Pro, Windows 2000 Pro                        |
| 192.168.10.133  | 2003-11-26 13:25:31 | Windows 2000                                            |
+-----------------+---------------------+---------------------------------------------------------+
17 rows in set (0.00 sec)
```

Table 1: This query and the corresponding results list all detected IP addresses and their corresponding operating systems. Here we also requested the date and time at which the operating system was last seen.

```
mysql> select ip_addr_dot,port, description, version from version where description = 'HTTP-S';
+---------------+------+-------------+-----------------------------------------------------------+
| ip_addr_dot   | port | description | version                                                   |
+---------------+------+-------------+-----------------------------------------------------------+
| 192.168.10.11 |   80 | HTTP-S      | Server: GWS/2.1                                           |
| 192.168.10.11 |   80 | HTTP-S      | Server: SmallWebServer/2.0                                |
| 192.168.10.10 |   80 | HTTP-S      | Server: Apache/1.3.28 (Unix)                              |
| 192.168.10.10 |   80 | HTTP-S      | Server: Apache/1.3.26 (Unix)                              |
| 192.168.10.10 |   80 | HTTP-S      | Server: GWS/2.1                                           |
| 192.168.10.10 |   80 | HTTP-S      | Server: SmallWebServer/2.0                                |
| 192.168.10.10 |   80 | HTTP-S      | Server: Microsoft-IIS/5.0                                 |
| 192.168.10.10 |   80 | HTTP-S      | Server: Barista/3.2.7.0005                                |
| 192.168.10.10 |   80 | HTTP-S      | Server: ValueAdExpress Server 2.0 UNIX (FreeBSD)          |
| 192.168.10.11 |   80 | HTTP-S      | Server: Squid/2.3.STABLE2                                 |
| 192.168.10.8  |   80 | HTTP-S      | Server: Squid/2.3.STABLE2                                 |
| 192.168.10.10 |   80 | HTTP-S      | Server: Stronghold/2.4.2 Apache/1.3.6 C2NetEU/2412 (Un ...|
| 192.168.10.8  |   80 | HTTP-S      | Server: GWS/2.1                                           |
| 192.168.10.8  |   80 | HTTP-S      | Server: Stronghold/2.4.2 Apache/1.3.6 C2NetEU/2412 (Un ...|
 (...)
| 192.168.10.8  |   80 | HTTP-S      | Server: Microsoft-IIS/4.0                                 |
| 192.168.10.8  |   80 | HTTP-S      | Server: Apache/1.3.27 (Unix) mod_throttle/3.1.2 mod_pe ...|
+---------------+------+-------------+-----------------------------------------------------------+
58 rows in set (0.00 sec)
```

Table 2: This query and the corresponding results list all Web servers seen on the local network. Web servers are recorded as “HTTP-S” to differentiate them from Web clients (“HTTP-C”).

"recently" (where "recently" is configurable but defaults to five minutes). In that case the Sniffer does not immediately update the database. The new most-recent time information is cached, and the database is updated later, either by a housekeeping routine or when the Sniffer exits. This caching means that the information in the database will not be as up-to-date as it would be without caching, but the performance increase is substantial. In concrete terms, without this caching, NetState was not able to keep up with the network traffic on our target network (averaging ˜7 Mb/s combined inbound and outbound traffic). With the caching, dropped packets were essentially reduced to zero as reported by `pcap_stats()`.

5 Experiences

After running NetState on our internal network for several months, we have already found some useful results. Mainly, NetState is useful for finding out what is really happening on the network and for spotting unusual activity that might not be detected by active scanning. For example, if multiple machines are located behind a NAT (network address translation) device, they will appear to have a single IP address. By monitoring the OS and application versions coming from that IP address, it is easy to detect a NAT device (or a single machine that boots multiple operating systems at different times). This sort of information is useful both because it might be in violation of network security policies and because we might want to identify all machines running a certain OS for patching and vulnerability assessment/remediation. NetState can also detect information about a machine that is used infrequently – such a machine might not even be turned on when an active scanning program is run, but if it is ever booted and communicates on the network, NetState can detect it.

Because NetState does not rely on "known ports" to identify application versions, it can detect services running on unusual ports. These might be unsanctioned HTTP servers, or they could be indications of a compromised machine "phoning home" to the attacker. Active scanning, obviously, can only detect the ports that happen to be open at the time the scan is performed. Attackers often only open ports for very short windows of time. Again, NetState can detect and log this activity whenever it happens to occur.

6 Challenges

As in any project, we were presented with some challenges in the course of our implementation. One challenge was designing a system that could handle tracking application versions over a very large IP address space (i.e. CIDR /16 and larger spaces). This required a large database with capability to hold information on, potentially, thousands of addresses and ports and, sometimes, multiple services per port corresponding to a single IP address because of multiple installed operating systems.

Another difficulty is application version obfuscation. Some network services issue version strings with varying degrees of specificity. Some services do not issue version strings at all, leaving version identification to a process of identifying protocol differences between versions. We do not currently use this technique for application version identification in NetState.

NetState cannot account for the situation created when the Sniffers are located on the outside of a NAT device. The NAT device causes many service versions to appear as if they are associated with one IP address or computer, creating many collisions in the NetState database. Many service versions for one given port can be recorded in a very short period of time causing an administrator great confusion. The solution to this situation, of course, is to design the monitoring architecture in such a way that a NetState Sniffer is behind every NAT device. Knowing where NAT devices are located on one's network is, of course, the most important help for an administrator. This same issue exists when detecting Web browser versions for machines behind a Web proxy server. As mentioned in Section 5 above, this aberrant behavior can be helpful in detecting NAT devices and Web proxy servers on networks, especially when these devices need to be regulated by an administrator in some way.

7 Legal Issues

Some network users may object to software such as NetState because it is a form of monitoring software and has the potential to invade one's privacy. We can appreciate that opinion and can assure users that NetState evaluates and stores data exactly as described previously and does not store data from further down in the data stream. Due diligence requires us to look at the legality of one's corporation, university, or ISP (Internet Service Provider) conducting such "monitoring" as well.

Corporations (such as our laboratory) are legally allowed to monitor their own networks for "business purposes" which could include monitoring for misuse and potential vulnerabilities [10, 11, 12]. In addition, we use banners in local login windows and remote logins to indicate that all network traffic is subject to monitoring. Logging into the system indicates consent to monitoring, though consent is not essential for a company to monitor employee communications. In almost all cases, employees should have no expectation of privacy relating to their network traffic including email (whether a work account or a personal account), Web-surfing habits, etc. [13]. When using company-owned equipment to access a data network, all network traffic is fair game for corporate snooping.

According to US code, ISPs are allowed to monitor their networks for misuse and potential vulnerabilities as well [14]. An ISP is allowed to "intercept, disclose, or use" the network traffic for the purposes of rendering service and for protecting its property. It can easily be seen that a system used for tracking service versions and thus potential vulnerabilities on an ISP's network, though not on systems owned by the ISP, can be used to ensure a properly functioning network for customers and protecting the ISP's own assets (servers, bandwidth, etc.).

It appears that universities can also sniff network traffic under the same US code section as above. Since most universities provide a "wire or electronic communications service," they can also protect their property using a tool such as NetState.

Employees, customers, students, researchers, etc. may not like that their Internet communications can be watched, but US law appears to allow such actions. Again, our tool does nothing more than watch for and record network service version information. Nevertheless, we remind users to deploy encrypted network applications or to tunnel their applications over an encrypted link for true data confidentiality.

8 Future Work

We would like to extend NetState to detect version strings for more network services. Eventually we would hope to have a list containing regular expression-based signatures for version strings so that we can easily add more detection capability. This could be similar to the signature file used by the open source intrusion detection system, Snort.

As mentioned earlier, the Nmap scanning tool has the capability to actively probe open ports for service and version information. It would be easy to quickly populate the NetState database upon initial installation using this feature of Nmap. NetState could piggyback on an organization's routine scanning activities to aid in database population as well.

Because NetState learns about service versions passively, it cannot learn information about specific software versions being run inside of SSL connections. Nmap invokes OpenSSL when it discovers an SSL-enabled service and then initiates further probes to obtain version information. We may add a module to NetState that invokes Nmap when SSL-enabled ports are discovered, storing those results in the NetState database.

NetState is a query-based tool. In other words, a network/security analyst will not get information out of NetState if he/she does not specifically ask for it. We would like to build a small set of signatures that constantly look for service version anomalies and automatically notify appropriate personnel. For example, we would like to know in a short amount of time if an OpenSSH service version changed to an earlier version than was last known.

Our current design using passive sniffing could aid in performing network-based anomaly detection in the future. Since the anomaly detection data would come from current traffic that was scanned passively, it can be directly compared to the data from NetState – i.e. the data will contain the same type of information. We think this will make the anomaly detection task more tractable.

We have begun to study creating network profiles for each device on our network using NetState. Because we have Sniffers placed in many strategic locations, it is easy to record and store information about the typical network traffic patterns seen from each network device. We have experimented with storing information about each session that a device establishes that terminates with hosts outside our networks. After enough time building a database of session data, we hope to extend the NetState Server so that it detects anomalies in network traffic between hosts.

9 Acknowledgements

We would like to thank Tim Toole, Tristan Weir, Archer Batcheller, Kami Vaniea, and Eric Thomas for their contributions and insight into this project. Special thanks goes to Randy McClelland-Bane for his hard work gathering screenshots and data for our consumption.

References

[1] Fyodor. (2005) Nmap. Insecure.org. [Online]. Available: http://www.insecure.org/nmap/

[2] eEye Digital Security. (2005) Iris network traffic analyzer. [Online]. Available: http://www.eeye.com/html/products/iris/index.html

[3] Javvin Company. (2005) Network packet analyzer. [Online]. Available: http://www.javvin.com/packet.html

[4] Novell Inc. (2005) Zenworks suite. [Online]. Available: http://www.novell.com/products/zenworks/

[5] Renaud Deraison. (2005) Nessus open source vulnerability scanner project. Tenable Network Security. [Online]. Available: http://www.nessus.org/

[6] Lawrence Berkeley National Laboratory Network Research Group. (2005) libpcap. [Online]. Available: http://ftp.ee.lbl.gov/nrg.html

[7] Michal Zalewski. (2005) p0f v2. [Online]. Available: http://lcamtuf.coredump.cx/p0f.shtml

[8] Darpa intrusion detection evaluation data. MIT Lincoln Laboratory. [Online]. Available: http://www.ll.mit.edu/IST/ideval/data/1999/training/week3/

[9] Aaron Turner and Matt Bing. (2005) Tcpreplay. [Online]. Available: http://tcpreplay.sourceforge.net/

[10] (2004) Workplace privacy. EPIC. [Online]. Available: http://www.epic.org/privacy/workplace/

[11] (2002) Fact sheet 7: Workplace privacy. Privacy Rights Clearinghouse. [Online]. Available: http://www.privacyrights.org/fs/fs7-work.htm

[12] Karen L. Casser, "Employers, employees, e-mail and the internet," in *The Internet and Business: A Lawyer's Guide to the Emerging Legal Issues*. Computer Law Association, 1996. [Online]. Available: http://www.cla.org/RuhBook/chp6.htm

[13] (2003) Fact sheet 18: Privacy in cyberspace. Privacy Rights Clearinghouse. [Online]. Available: http://www.privacyrights.org/fs/fs18-cyb.htm

[14] United States Federal Government, "Interception and disclosure of wire, oral, or electronic communications prohibited," in *US Code, Title 18, Part I, Chapter 119, §2511*, 2004. [Online]. Available: http://www.law.cornell.edu/uscode/html/uscode18/usc_sec_18_00002511----000-.html

OpenCSG: A Library for Image-Based CSG Rendering

Florian Kirsch, Jürgen Döllner

University of Potsdam, Hasso-Plattner-Institute, Germany
`{ florian.kirsch, juergen.doellner} @hpi.uni-potsdam.de`

Abstract

We present the design and implementation of a real-time 3D graphics library for image-based Constructive Solid Geometry (CSG). This major approach of 3D modeling has not been supported by real-time computer graphics until recently. We explain two essential image-based CSG rendering algorithms, and we introduce an API that provides a compact access to their complex functionality and implementation. As an important feature, the CSG library seamlessly integrates application-defined 3D shapes as primitives of CSG operations to ensure high adaptability and openness. We also outline optimization techniques to improve the performance in the case of complex CSG models. A number of use cases demonstrate potential applications of the library.

1 Introduction

Constructive Solid Geometry (CSG) represents a powerful 3D modeling technique. The idea of CSG is to combine simple 3D shapes to more complex ones with Boolean operations in 3-dimensional space. Even though CSG is an established technique and it is well-understood, it has not become mainstream in software systems due to the complex implementations of rendering algorithms to display CSG models.

In the scope of CSG, the most basic shapes are called primitives. A CSG primitive must be solid, i.e., given in a way that interior and exterior regions of the primitive are clearly defined. For example, a sphere or a cube is a solid primitive, whereas a triangle is not. Two primitives or CSG shapes can be combined by one of the following Boolean operations to define a more complex CSG shape:

- **Union**. The resulting shape consists of all regions either in the first, in the second, or in both input shapes.
- **Intersection**. The resulting shape is the region common to both input shapes.
- **Subtraction**. The resulting shape is the region of the first shape, reduced by the region of the second shape.

CSG models are stored in CSG trees, where leaf nodes contain primitives and inner nodes contain Boolean operations (Figure 1). Because of the mathematical properties of Boolean operations, the resulting CSG shapes are always solid. This is an important advantage

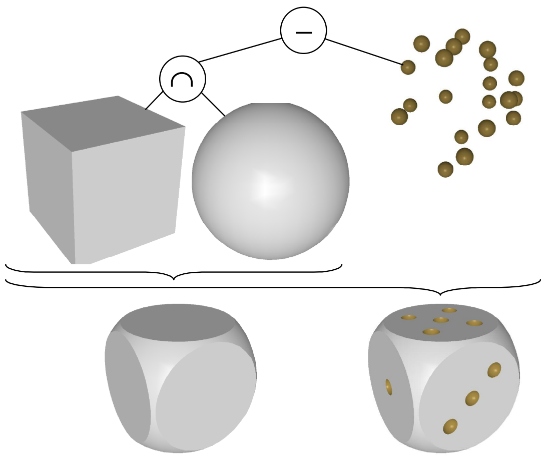


Figure 1. Modeling a dice using CSG. A cube and a sphere are intersected; from the result the dots of the dice are subtracted.

over other 3D modeling techniques, which often miss polygons and generate unclosed models.

CSG modeling and rendering is directly available in several graphics systems for offline rendering, such as POV-Ray [1] or RenderMan [2]. Those graphics systems are used to generate photo-realistic images, and they are not suited for real-time rendering, though.

Rendering CSG shapes in real-time using and taking advantage of graphics hardware is difficult, in particular if the CSG shape is modified interactively. Basically, there are two options: On the one hand, the boundary of the CSG shape can be calculated mathematically and stored in a polygonal model that then is sent to graphics hardware. This is practical for static CSG shapes, but for CSG shapes that are modified interactively, the expensive calculation of the boundary must be repeated for each rendering frame, forbidding animated real-time display.

On the other hand, *image-based CSG rendering algorithms* can determine and store the visible parts of the CSG shape directly in the frame buffer of the graphics hardware. The result of an image-based CSG algorithm is, therefore, just the image of the CSG shape. Based on recent advances of graphics hardware images of CSG models can be generated instantaneously and, for models of considerable complexity, in real-time.

Image-based CSG can be used in all situations where complex 3D modeling operations are required in real-time. Image-based CSG does not analytically calculate the geometry of 3D objects, but most uses of CSG do not require the explicit 3D geometry and are satisfied with the image of the model. Still, image-based CSG is not commonly used in real-world applications today. Very often, the primitives of CSG shapes are only sketched as wire-frame image, which leads to difficult understanding of the final 3D-shapes' look.

We have developed OpenCSG, which represents the first free library for image-based CSG rendering. OpenCSG is written in C++, bases on OpenGL, and implements the two most important image-based CSG algorithms: The Goldfeather algorithm, which is suited for all kinds of CSG primitives, and the SCS algorithm, which is a more optimal algorithm if a CSG shape is composed of only convex CSG primitives.

2 Related Work

Constructive Solid Geometry (CSG) has been recognized as powerful approach for modeling complex 3D geometry for a long time [3]. The foundation of image-based CSG rendering was invented by Goldfeather et al. [4]. They described the normalization of CSG trees and also developed the first implementation of an image-based CSG algorithm. Another important class of CSG rendering algorithm today is the SCS algorithm [5].

2.1 Normalization of CSG Trees

Rendering arbitrary CSG trees directly in real-time is still not possible today. Instead, a CSG tree must first be normalized. A *normalized* CSG tree is in sum-of-products form, i.e., it is the union of several *CSG products* that consist, respectively, of a CSG tree with intersection and subtraction operations only, and only one single primitive is allowed to be the second operand of each operation. In other words, a CSG product has the form $(\ldots(x_1 \otimes x_2) \otimes x_3) \ldots \otimes x_n)$ where "$\otimes$" is either an intersection or a subtraction. Goldfeather et al. proved that the following set of equivalences, when applied repeatedly to inner nodes of an arbitrary CSG tree, transform the tree to a normalized CSG tree:

1. $x - (y \cup z) \rightarrow (x - y) - z$
2. $x \cap (y \cup z) \rightarrow (x \cap y) \cup (x \cap z)$
3. $x - (y \cap z) \rightarrow (x - y) \cup (x - z)$
4. $x \cap (y \cap z) \rightarrow (x \cap y) \cap z$
5. $x - (y - z) \rightarrow (x - y) \cup (x \cap z)$
6. $x \cap (y - z) \rightarrow (x \cap y) - z$
7. $(x - y) \cap z \rightarrow (x \cap z) - y$
8. $(x \cup y) - z \rightarrow (x - z) \cup (y - z)$
9. $(x \cup y) \cap z \rightarrow (x \cap z) \cup (y \cap z)$

Both the Goldfeather and the SCS algorithm render one CSG product at a time. The union of several products is determined by normal use of the depth buffer.

2.2 The Goldfeather Algorithm

The Goldfeather algorithm, for the case of convex primitives, bases on several observations (Figure 2): First, only the front face of intersected and the back face of subtracted primitives are potentially visible. Second, each other primitive *P* affects the visibility of a pixel in a potentially visible polygon by the number of polygons of *P* in front of the pixel (called the *parity*, which, for convex primitives, is always in-between zero and two): If *P* is subtracted and the parity is one, the pixel is not visible, and if *P* is intersected and the parity is zero or two, the pixel is also not visible. In all other cases, the pixel could be visible, as at least primitive *P* does not affect its visibility.

Based on these observations, the Goldfeather algorithm works as follows: It tests the visibility of each primitive in a CSG product separately. For this, the front respectively back face of the primitive is rendered into a temporary, empty depth buffer. Then, for all other primitives in the CSG product, the parity is determined, i.e., the number of polygons of the primitive in front of the depth buffer is counted. On modern graphics hardware, this is possible by toggling a bit in the stencil buffer [6], an additional kind of depth buffer that supports Boolean operations. Finally, if no parity

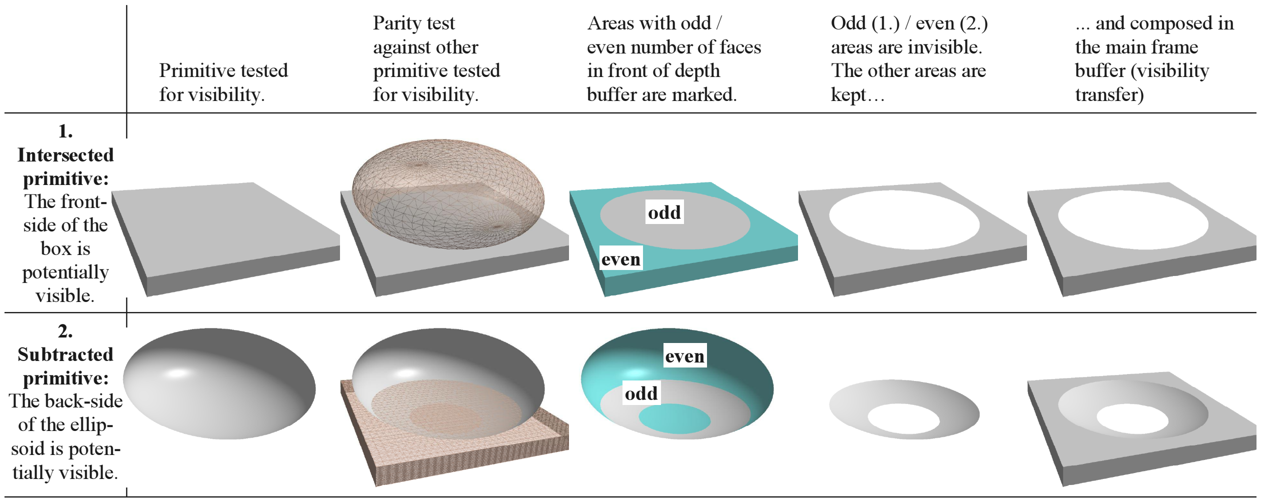

Primitive tested for visibility.
Parity test against other primitive tested for visibility.
Areas with odd / even number of faces in front of depth buffer are marked.
Odd (1.) / even (2.) areas are invisible. The other areas are kept…
... and composed in the main frame buffer (visibility transfer)
1. Intersected primitive: The front-side of the box is potentially visible.
odd
even
2. Subtracted primitive: The back-side of the ellipsoid is potentially visible.
even
odd

Figure 2. Example of the Goldfeather algorithm. An ellipsoid is subtracted from a box. The two primitives are tested for visibility separately. For each primitive, this requires calculating and analyzing the parity. When visible areas of a primitive have been determined, they are combined with the former content in the main frame buffer.

indicates that a pixel is invisible due to other primitives, the pixel is marked as visible, whereas the depth values of all other pixels are reset. The temporary depth buffer then is combined with the content of the main depth buffer before the algorithm continues with testing the visibility of the next primitive.

The basic Goldfeather algorithm can be used with a limited set of primitives, but some refinements are required to use the algorithm in more general cases:

Support of concave primitives [4]. First, the visibility of each potentially visible layer of a primitive must be checked separately. Layers of primitives can be rendered with the stencil test. Second, the parity calculation must be extended to distinguish an odd from an even number of polygons in front of a pixel. In practice, this is still possible by toggling a bit in the stencil buffer.

Resolution of the stencil buffer [6]. The stencil buffer typically only holds 8 bits for each pixel, such that it can only store the result of 8 parity tests at once. If more primitives are contained in a CSG product, invisible pixels must be remembered as such before continuing with the calculation of the remaining parities.

The following improvements and optimizations drastically increase rendering performance and, for complex CSG shapes, make image-based CSG rendering usable in practice:

Emulating two depth buffers. The Goldfeather algorithm requires two depth buffers, whereas graphics hardware directly exposes only one. This requires extensive copying between the depth buffer and main memory to emulate two depth buffers [6], or (far better) using an additional offscreen rendering canvas that holds the temporary depth buffer [7].

Visibility transfer. To combine the content of the temporary depth buffer with the main depth buffer, the depth values could be just copied with pixel operations [6]. But this requires a round-trip of the depth values to main memory. It is, therefore, faster by far to use a texture-based visibility transfer that can work completely on the graphics hardware [7].

Depth complexity. The basic Goldfeather algorithm has $O(n^2)$ run-time complexity, where n is the number of primitives in a CSG product. For large CSG products, it is often better to test the visibility of a complete depth layer at once [8]. This leads to a run-time complexity of $O(n \cdot k)$, where k is the depth complexity of the CSG product. The depth complexity can be calculated either using the stencil test or with occlusion queries [9].

Object-space optimizations. Primitives in a CSG product that do not overlap in screen space do not modify the visibility of one another. Therefore, the visibility of such primitives can be tested at once, and also the mutual calculation of the parity is not required (Section 6.3).

Image-space optimizations. Visible parts of a CSG product can only be inside the intersection of the bounding boxes of all intersected primitives in screen-space. The remaining areas do not need to be considered at all and can be omitted, for example, with the scissor test (Section 6.3).

2.3 The SCS Algorithm

The SCS Algorithm is optimized for convex primitives. In practical cases, it is typically faster than the Goldfeather algorithm, but in exchange it does not operate on concave primitives.

The SCS algorithm, similar to the Goldfeather algorithm, requires a normalized CSG tree. Its performance advantage is due to the determination of depth values of a complete CSG product in a temporary depth buffer at once. The algorithm has three stages: First, it determines the backmost front-face of all intersected primitives in the CSG product. Then, it subtracts primitives, i.e., where the front-face of a subtracted primitive would be visible and the back-face not, it replaces values in the depth buffer with the back-face depths of the subtracted primitive. For each subtracted primitive in a CSG product, this has to be done several times. Finally, the algorithm clips the depth buffer with the back-faces of all intersected primitives in the CSG product.

The SCS algorithm is subject to similar performance optimizations as the Goldfeather algorithm: It uses two depth buffers and requires a similar visibility transfer. It can also profit from knowing the depth complexity of a CSG product and object-space respectively image-space optimizations can be applied in a similar way.

2.4 Further Algorithms

Further image-based CSG algorithms appear to use two or more depth tests at the same time for rendering [10]. For a long time, standard graphics hardware did not support this. Guha et al. use the shadow mapping capability of modern graphics hardware for two-sided depth tests to implement their CSG algorithm [11]. This algorithm represents no fundamental new CSG rendering paradigm; It could be integrated into OpenCSG once hardware support matures.

2.5 Image-Based CSG Libraries

The only library for image-based CSG we are aware of is TGS SolidViz [12]. As this library is part of TGS OpenInventor 5.0, applications that are not based on OpenInventor, a rather large scene-graph library, cannot deploy SolidViz easily. TGS OpenInventor is available under a commercial license, the source code of SolidViz is not freely available, and the capabilities and restrictions of SolidViz are hardly documented.

Other implementations for real-time CSG rendering appear to be prove-of-concepts, which are demonstrated with small example programs. Therefore, they can be hardly integrated into other applications.

3 Overview of the OpenCSG Library

In the following, we describe our approach for the library for image-based CSG rendering. We will motivate the design choices in this section before we describe the consequences for implementation and usage of the library.

We consider the following points as important properties of a library for image-based CSG rendering:

1. A minimal, abstract, and well-defined interface, for easy use of the library. OpenCSG does not assume any specific type of CSG implementation.

2. A simple and well-defined output. The library is solely used for CSG rendering. All other tasks such as shading the CSG primitives are handled outside of the library.

3. Direct applicability to all kinds of rendering applications. Most graphics applications define their own set of graphical primitives, which are likely valid CSG primitives. It must be possible to use these graphical primitives for CSG rendering.

4. As few external dependencies as possible. A library that depends on a full-featured scene-graph library is clearly not acceptable.

5. Stability, performance, and portability as properties that every (graphics) library should provide.

OpenGL is the rendering library of choice to create portable real-time graphics applications that take advantage of the graphics hardware [13]; therefore, we use it for OpenCSG. Additionally, we require two small libraries: GLEW [14] is a library which manages loading and using OpenGL extensions, which provide a mean for using new rendering functionality that has not (yet) been adopted by the core OpenGL library. RenderTexture [15] is another small library that provides platform-independent access for hardware-accelerated offscreen rendering into textures by the means of p-buffers [16], a rendering technique that is extensively used by OpenCSG. Both GLEW and RenderTexture are, currently, statically linked with OpenCSG.

The most portable programming language for implementing a rendering library is C. However, the API of the library can be greatly simplified by using an object-oriented language such as C++ instead of C, especially for supporting user-defined primitives (design goal 3): In C, the rendering function of a CSG primitive would be specified using a function pointer. C++, as language with polymorphic objects, allows the same in a more convenient way by declaring an abstract rendering method in a primitive base class and implementing this method in derived classes. For this reason, OpenCSG is implemented in C++ instead of C.

4 The API of OpenCSG

In this section, we shortly describe the complete API of OpenCSG. The API is very compact and consists of only one class for CSG primitives and a rendering function that has a list of CSG primitives as argument and renders the CSG product indicated by this list of primitives into the depth buffer.

All classes, functions, and enumerations of OpenCSG are members of the C++ namespace `OpenCSG`.

4.1 Specifying Primitives

The interface of OpenCSG defines an abstract base class for all kinds of CSG primitives. Primitive objects can be assigned a bounding box in normalized device coordinates (screen-space mapped to $[-1, 1]^2$), which is used for object-based and image-based rendering optimizations internally. Setting the bounding box is optional. Primitives also have a convexity, i.e., the maximum number of depth layers of the primitive. The developer must set the convexity because some algorithms such as the Goldfeather algorithm need to know the convexity of a primitive for correct rendering operation. The convexity also allows for choosing between different CSG rendering algorithms that are provided by OpenCSG

The render method of the CSG primitive class is abstract. To use OpenCSG the developer must derive a specialized primitive class, which implements the render method. The CSG rendering function requires that the geometric transformation used for rendering a CSG primitive does not depend on transformations of other primitives. This is best done by pushing the transformation matrix at the beginning of the primitive's render function, and restoring the matrix from the transformation stack at the end. An alternative approach is to load the correct transformation matrix unconditionally for all primitives in a CSG product at the beginning of the primitive's render functions.

For internal use in OpenCSG, the render method of a primitive must not change the primary color. The color of a primitive is used by all CSG rendering algorithms for the texture-based visibility transfer, hence altering it in the rendering method causes invalid rendering results. For best rendering performance vertex positions alone should be submitted to graphics hardware because only pure geometry is needed for correct operation of CSG rendering. All other per-vertex data such as normals or texture coordinates is ignored.

```
namespace OpenCSG {
    enum Operation { Intersection, Subtraction };
    class Primitive {
    public:
        Primitive(Operation, unsigned int convexity);
        virtual ~Primitive();
        void setOperation(Operation);
        Operation getOperation() const;
        void setConvexity(unsigned int);
        unsigned int getConvexity() const;
        void setBoundingBox(float  minx, float  miny, float  minz,
                            float  maxx, float  maxy, float  maxz);
        void getBoundingBox(float& minx, float& miny, float& minz,
                            float& maxx, float& maxy, float& maxz) const;
        virtual void render() = 0;
    };
    enum Algorithm {
        Automatic, Goldfeather, SCS
    };
    enum DepthComplexityAlgorithm {
        NoDepthComplexitySampling, OcclusionQuery, DepthComplexitySampling
    };
    void render(const std::vector<Primitive*>& primitives,
                Algorithm = Automatic,
                DepthComplexityAlgorithm = NoDepthComplexitySampling);
}
```

Figure 3. The API of OpenCSG

4.2 Rendering CSG Models

The central part of the OpenCSG library are the CSG rendering algorithms. For using them, the API of OpenCSG defines a rendering function, which requires an argument containing the CSG shape to render. During the design of the API we considered two different options for that argument: It could be a full-featured CSG tree or only a CSG product, i.e., a list of primitives, each of them either subtracted or intersected. In the following, we discuss these two approaches.

First consider the required steps for the user of OpenCSG if the argument of the CSG rendering function would be a CSG tree. In this case the application developer would have to compose the CSG tree in its application. This would require a class for CSG primitives and some node classes for the inner nodes of the CSG tree, i.e., union, intersection, and subtraction. While this would not be particularly difficult, it would nonetheless require a complete API for composing and modifying CSG trees.

The second option is to permit only a CSG product as argument of the rendering function. In this case the interface can be drastically reduced: The rendering function only requires a list of CSG primitives that can be just given as STL `vector`. Additionally each CSG primitive must hold a flag to indicate whether it is subtracted or intersected.

If the argument of the CSG rendering function is a complete CSG tree, the rendering library must also implement the normalization of the CSG tree into a set of CSG products. In the other case, this normalization step has to be potentially implemented by the application developer. But implementing the normalization itself is not particularly difficult. Also, an application can often create a normalized CSG tree of smaller size than a general function in a CSG library because it can additionally analyze application-specific information. Furthermore, if the rendering function supported an arbitrary CSG tree as argument, the application might nonetheless require a non-trivial transformation of its CSG data to the CSG tree argument of the rendering function, depending on the kind of Boolean operations supported by the application.

Therefore, the render function of OpenCSG takes a CSG product as argument, given as a list of CSG primitives. We assume general functionality for normalizing a CSG tree would be better implemented by a separate library.

Furthermore, the render function has two optional arguments to choose between different CSG algorithms. The first of these arguments chooses between the Goldfeather algorithm, the SCS algorithm, or an automatic mode that chooses between both algorithms depending on the primitives in the CSG product. The second of these arguments chooses between different strategies for analyzing the depth complexity of the CSG product: The depth complexity can be ignored, determined directly by counting depth layers in the stencil buffer, or used indirectly with occlusion queries.

5 Output Generated by OpenCSG

The CSG rendering algorithms in OpenCSG work all in a similar way. From perspective of their rendering result they are exchangeable: They initialize the depth information in the depth buffer with the correct depth values of the CSG product. The depth values are written with respect to the depth values that were in the depth buffer before the CSG rendering call, using the common z-less depth function. This allows for scene composition of different CSG products with correct hidden surface removal.

Besides altering the depth buffer, the CSG rendering algorithms try to avoid any side effect. For example, OpenCSG does not alter the color buffer, i.e., it does not shade the colors of CSG shapes. This is because of the many different approaches and techniques for color shading. A number of graphics libraries are concerned with the task to do color shading well. There are even algorithms such as shadow mapping that do not require color shading of CSG primitives at all. Therefore, shading the primitives after initializing the depth buffer is better handled by the application.

The stencil buffer as remaining important part of the frame buffer should also not be affected by using the CSG rendering function. Currently, this is a technical problem for CSG shapes that contain concave primitives. During the visibility transfer, the CSG rendering function must render separate depth layers of these primitives in the frame buffer. This functionality is implemented by counting the depth layers in the stencil buffer, thereby destroying its former content. Hence, the CSG rendering function guaranties to preserve the stencil buffer only in the case of convex primitives, otherwise the content of the stencil buffer is undefined.

The CSG rendering function respects a number of settings of the OpenGL state for CSG rendering, whereas it ignores other settings that are required internally or simply do not make sense: Above all, the numerous OpenGL settings that manipulate color calculation are irrelevant for the CSG rendering function, which does not alter the color buffer. The meaning of other important OpenGL settings for CSG rendering with OpenCSG is summarized in the following.

Alpha Test. OpenCSG internally uses the alpha test for visibility transfer. Therefore the application settings of the alpha test are ignored.

Scissor Test. This fragment test is applied as usual, i.e., fragments are only generated inside of the scissoring area.

Stencil Test. The stencil test will be performed correctly if only convex primitives are contained in the CSG product and if no layered CSG algorithms is used. In the other cases, our algorithms uses the stencil test internally and also overwrite the stencil buffer of the frame buffer.

Depth Test. Depth-test settings are completely managed by OpenCSG. It renders with the z-less depth function, i.e., the forefront of the CSG shape is rendered where it is in front of the former depth value in the depth buffer. The z-greater function would be the only potential alternative: The effect would be to render the backside of the CSG shape behind the former depth value in the depth buffer. Since the benefit of this setting is questionable and since it requires large changes to the implementation of the internal CSG algorithm, we did not implement this.

Culling. The setting of front- and back-face culling is ignored because OpenCSG uses this setting internally. Anyway, culling also lacks a sound meaning for CSG rendering of possibly concave CSG shapes.

Geometric Transformations. OpenCSG does not change the transformation and projection internally. It is under the responsibility of the CSG primitives to set and restore the necessary transformations.

6 Implementation Details

The implementation of the CSG algorithms in OpenCSG is similar to the algorithms presented by Kirsch and Döllner [7], and it uses many of the techniques presented there to improve rendering performance. Additionally, OpenCSG performs both object-based and image-based performance optimizations.

6.1 Choosing the CSG Algorithm

The CSG rendering function for rendering a CSG product has two parameters that control what kind of algorithm is used for CSG rendering. As basic algorithms, the Goldfeather algorithm and the SCS algorithm are provided. Each of them can by used in several variants: The depth complexity of the CSG product can be determined (1) by counting the overdraw of the CSG product in the stencil buffer, (2) indirectly by the means of occlusion queries, or (3) not at all.

OpenCSG users can also specify the automatic mode, in which an internal heuristic guesses the fastest algorithm for rendering the CSG product. The following heuristic experimentally turned out to give satisfactory results: The SCS algorithm is chosen if the CSG product contains only convex primitives, else the Goldfeather algorithm is used. As strategy for depth complexity, hardware occlusion queries are used if the CSG product contains more than 20 primitives and the hardware supports them. If the hardware does not support them and the CSG product contains even more than 40 primitives, depth complexity is calculated by counting the overdraw of the CSG product with the stencil buffer. In all other cases, the depth complexity is not determined at all.

6.2 P-Buffer Settings

For supporting two depth buffers, OpenCSG uses p-buffers, i.e., offscreen rendering canvases. The visibility transfer from temporary to main depth buffer is performed using RGBA-textures, i.e., a texture is created in which each color channel encodes the visibility of a CSG primitive (Goldfeather) respectively the alpha channel encodes one CSG product (SCS). This texture is used to reconstruct the depth values in the main depth buffer.

The p-buffer for internal calculation of the primitive's visibility must be configured carefully such that it behaves identically to the main frame-buffer. This is required because p-buffers have their own rendering context with rendering settings that are independent of the main rendering context. So we replicate the viewport-size, the transformation and projection matrix, and the front-face setting.

Determining an optimal physical size of the p-buffer is not trivial. First we tried the approach to set the size of the p-buffer to the size of the frame buffer in the calling application (or to the next power-of-two size in case of missing support for non-power-of-two-textures). But this approach does not work well for applications using two or more frame buffers displaying CSG shapes. As the sizes of the frame buffers likely differ the p-buffer is constantly resized resulting in unacceptable rendering performance. This does not only

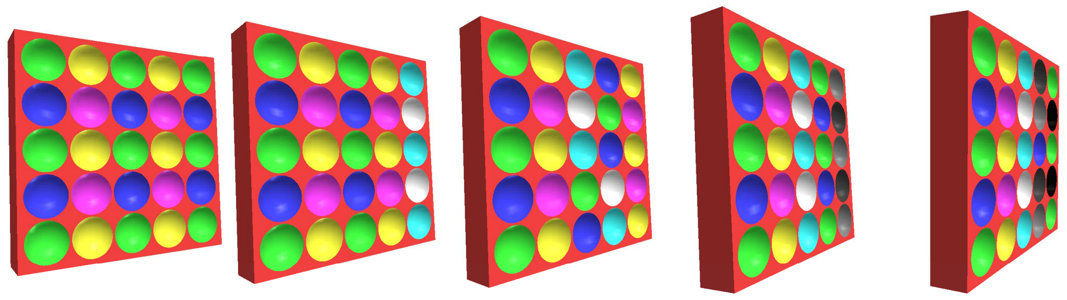


Figure 4: Primitive Batches. In these images, primitives that are contained in a batch are depicted with the same color. The number of primitive batches depends of the orientation of the CSG product with respect to the camera: In the left image, there are six primitive batches (included the box); in the right image there are twelve.

occur in applications with several OpenGL-windows; it is also an issue for multipass algorithms such as dynamic shadow mapping or reflections by the means of textures because those algorithms internally require additional frame buffers.

We solved this problem with a heuristic for resizing the p-buffer consisting of two rules: If the size of the current p-buffer is smaller than the requested size $x_0 \times y_0$, the p-buffer is enlarged to that size immediately. Otherwise we analyze the requested sizes in both dimensions during the last N requests $x_0 \dots x_N$ and $y_0 \dots y_N$ and calculate the respective maxima $x_{max} = \max \{x_i \mid i \in 0 \dots N\}$ and $y_{max} = \max \{y_i \mid i \in 0 \dots N\}$. If for a request the size of the current p-buffer exceeds $x_{max} \times y_{max}$, we downsize the p-buffer to the new size $x_{max} \times y_{max}$. In practice, a value of 100 for N proves to get good results: The p-buffer is always big enough, it is not resized too often, and memory for an unnecessary large p-buffer is not occupied forever.

6.3 Performance Optimizations

Technical publications about image-based CSG rarely mention the fact that the performance of CSG rendering can be drastically improved with rather simple optimizations that just take into account the bounding boxes of primitives in the CSG product. OpenCSG implements the following optimizations:

Primitive Batches. For the standard Goldfeather algorithm, normally the visibility of each primitive is calculated separately. But obviously, primitives that do not overlap in z-direction of the camera space do not mutually influence themselves. Therefore, primitives whose bounding boxes do not overlap in z-direction are internally grouped to batches of primitives (Figure 4). Further on, each batch is treated as a single primitive. This way we reduce the number of parity tests and also lower the number of copy operations from the temporary into the main depth buffer. The same idea is applied to the SCS algorithm with all depth-complexity strategies, whereas it makes no sense for the Goldfeather algorithm using depth-complexity sampling.

Disjoint Bounding Volumes. In the standard Goldfeather algorithm, calculating the parity of a subtracted primitive against a primitive batch is only required if its bounding box intersects at least one bounding box of a primitive in the primitive batch. Otherwise, the subtracted primitive cannot alter the visibility of the primitive batch anyway. This optimization does neither apply to the Goldfeather algorithm using depth-complexity sampling nor to the SCS algorithm.

Restrict Rendering to Intersection Area. Visible parts of the CSG product can only be found where intersected primitives overlap in the xy-plane of camera space. Hence, we calculate the bounding boxes of all intersected primitives in pixel coordinates, intersect them, and restrict rendering during the complete calculation of the CSG product to this *intersection area* using scissoring. This optimization is applicable to all image-based CSG algorithms.

Restrict Rendering to Primitive Batch. For the parity calculation in the standard Goldfeather algorithm, the scissoring area can be minimized even more: The parity does not need to be calculated for pixels outside of the bounding box of the primitive batch whose visibility is being determined because there are no visible pixels of the primitive batch anyway. Therefore, we set the scissor region such that the parity is only calculated in the intersection of the bounding box of the primitive batch and the aforementioned intersection area.

7 Using OpenCSG

In this section, we describe a small example for CSG rendering with OpenCSG in practice. We also give some practical hints for shading CSG primitives for more complex applications and lighting conditions.

7.1 Overview of the Rendering Process

Rendering CSG shapes with OpenCSG requires two steps: First, OpenCSG initializes the depth buffer of one or several CSG products. Then, the application is responsible for the color shading of the CSG primitives. For that, the primitives must be rendered using a z-equal depth function and with exactly the same transformations that are used in the CSG primitive objects. As only the front faces of intersected and the back faces of subtracted primitives can be visible, back or front face culling can be used accordingly to enhance rendering performance. The resulting performance improvements are, however, hardly noticeable. Culling is not important to get a correct rendering result, since the depth test for equality already filters out all invisible fragments.

7.2 A First Example

The OpenCSG library contains the example described in the following, which illustrates a simple but effective way to use it. The application creates OpenGL display lists for each CSG primitive, which are created using the OpenGL helper libraries GLU and GLUT.

```
GLuint id1 = glGenLists(1);
glNewList(id1, GL_COMPILE);
glutSolidCube(1.8);
glEndList();
GLuint id2 = glGenLists(1);
glNewList(id2, GL_COMPILE);
glutSolidSphere(1.2, 20, 20);
glEndList();
```

The application also derives a concrete class `DLPrim` for CSG primitives, whose render-method just executes one display-list id.

```
class DLPrim : public OpenCSG::Primitive {
public:
  DLPrim(unsigned int displayListId,
         OpenCSG::Operation operation,
         unsigned int convexity)
    : OpenCSG::Primitive(operation, convexity),
      id(displayListId) { }

  virtual void render() { glCallList(id); }

private:
  unsigned int id;
};
```

For each CSG primitive, an object of this class is created that holds the appropriate display list id for rendering. A CSG product as argument for the render function of OpenCSG is constructed just by appending several CSG primitives to an STL `vector` :

```
namespace OpenCSG;
DLPrim* box=new DLPrim(id1, Intersection, 1);
DLPrim* sphere=new DLPrim(id2,Subtraction, 1);
std::vector<Primitive*> primitives;
```

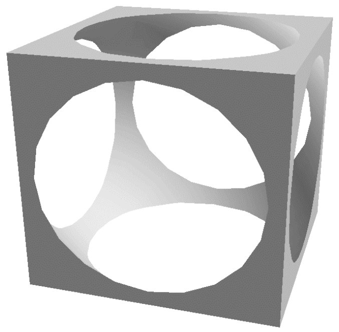


Figure 5. Resulting image of the CSG model defined in the example.

```
primitives.push_back(box);
primitives.push_back(sphere);
```

For rendering the CSG product described in this way, we invoke the render function with the list of primitives. Optionally, we can choose a specific algorithm and a strategy for depth complexity sampling.

```
OpenCSG::render(primitives,
                Goldfeather,
                NoDepthComplexitySampling);
```

After this, the depth buffer contains the correct depth values of the CSG product. Now the application is responsible for shading the CSG product with the desired colors. We set the depth comparison function to z-equal, such that only fragments are rendered that equal in depth to the value stored in the depth buffer. We also set up lighting and shading as desired and then render all primitives in the CSG product.

```
glDepthFunc(GL_EQUAL);
// setup lighting and shading
for (int i=0; i<primitives.size(); ++i) {
    (primitives[ i] )->render();
}
glDepthFunc(GL_LESS);
```

Finally, we reset depth testing to its original value. Figure 5 shows the image that is rendered by this example.

The bounding boxes of the primitives are not set in this example. This is fine for such a small CSG product with few optimization opportunities. But for CSG products that consist of more primitives setting the bounding boxes is likely to increase rendering performance.

7.3 Lighting CSG Shapes

For illuminating CSG shapes, it is important to distinguish between intersected and subtracted primitives: polygons of intersected primitives are oriented normally, but subtracted primitives have an inverse orientation because only their inner back-facing polygons can be visible. Therefore the lighting of subtracted primitives is, without precautions, inverse to the standard lighting: polygons that point away from the light are lighted, but polygons that are oriented towards the light source are unlighted.

The first option to fix this issue is to negate the normals of all subtracted primitives. This is likely the best option if vertex programs are used for geometrical transformations because a vertex program can implement the negation of normals easily. Otherwise this is a potentially more complicated operation, since the normals of primitives must be created analytically, respecting the CSG operation of the primitive. Negating normals is the only option that works correctly for shading by the means of per-pixel lighting respectively bump-mapping in a fragment program.

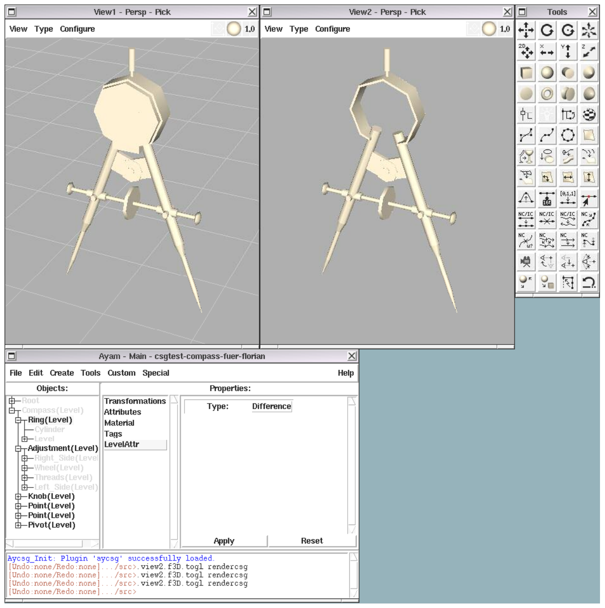
The second option for the standard vertex lighting is to enable two-sided lighting. In this case, vertices of back-facing polygons have their normals reversed before the lighting equation is evaluated. Unfortunately, some graphics hardware such as the GeForceFX takes a big performance hit with two-sided lighting. Apart from this performance weakness, this is a simple solution for correct lighting.

A third workaround for the lighting problem is to place two identical light sources on the opposite sides of the CSG model: one of them illuminates polygons of intersected and the other one polygons of subtracted primitives facing to the same direction. While this may be a kludge, in practice this approach works well if we can accept an illumination that is equal from both sides of the CSG model.

8 Limitations and Performance

As a general problem of image-based CSG rendering, CSG shapes are only correctly rendered if all primitives are completely inside the view frustum. Otherwise the internal calculations of the CSG algorithms, for example calculating the parity, fail. The fundamental problem is that image-based CSG rendering requires solid primitives, but if a solid primitive crosses the front- or back plane of the view frustum, some parts are clipped and the primitive is not closed, and, therefore, not solid in the view frustum anymore.

With the SCS algorithm, an alpha-texture encodes the visibility of all primitives in a CSG product. As the alpha channel typically has a resolution of 8 bits only, 256 different values can be distinguished in a channel, therefore the CSG products may only consist of at most



Type: Difference

```
Aycsg_Init: Plugin 'aycsg' successfully loaded.
[Undo:none/Redo:none].../src>.view2.f3D.togl rendercsg
[Undo:none/Redo:none].../src>.view2.f3D.togl rendercsg
[Undo:none/Redo:none].../src>.view2.f3D.togl rendercsg
[Undo:none/Redo:none].../src>
```

Figure 6. Screenshot of the RenderMan modeling environment Ayam using OpenCSG for previewing CSG models.

255 primitive batches. In our experience, this limitation can only be exceeded in pathological cases. As a future work, IDs could be spread over the whole RGBA channel, allowing $2^{32}-1$ different primitive batches for a CSG product.

CSG rendering with OpenCSG requires some OpenGL extensions. At least, a p-buffer must be available for offscreen rendering. Also, a stencil buffer must be supported in the p-buffer, and, for rendering concave shapes, also in the main frame-buffer. However, very old graphic hardware does not support stencil or p- buffers. In practice, OpenCSG can be used on all newer graphics hardware: The oldest NVidia graphics chip that is supported by OpenCSG is the Riva TNT, and the oldest ATI generation is probably the Rage 128 (the latter hardware was, till now, not available for testing, though).

For acceptable performance, using more recent graphics hardware is important, whereas the power of the CPU does not matter much. We have observed excellent performance on GPUs such as the GeForce4, GeForceFX, or ATI Radeon 9800. On such graphics hardware, OpenCSG can use additional hardware features such as hardware occlusion queries for depth-complexity sampling and dot products in the texture environment, which are used to improve the performance of the visibility transfer.

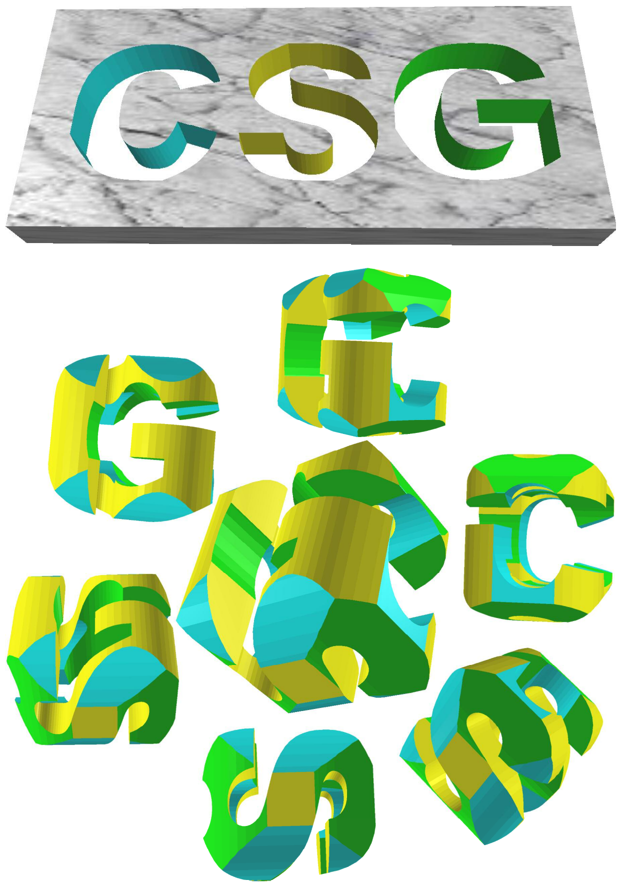


Figure 7. How does a 3D object look like that exactly fits through all letter-shaped holes? CSG gives the answer: Just intersect the extruded 3D-models of each letter. The resulting shape is shown below from different positions.

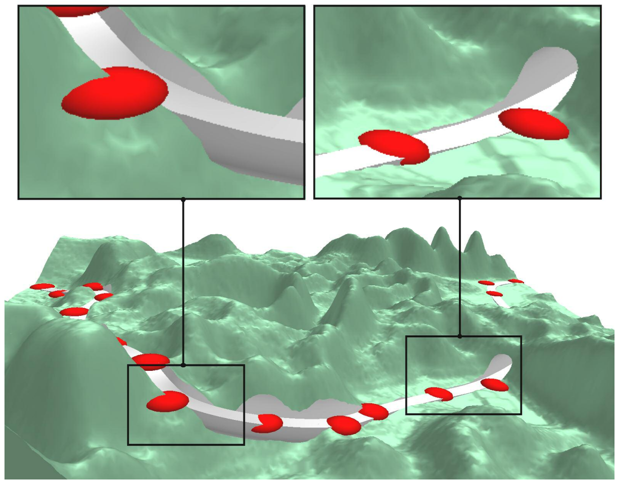


Figure 8. Interactive road design in a 3D terrain model using CSG operations to model excavations, embankments, and tunnels.

In practice, with an ATI 9800 and a window resolution of 800×600, the trivial examples in Figure 2, 4, and 5 achieve several hundred fps. The dice in Figure 1 renders at about 100 fps, the puzzle of Figure 7 at about 60 fps, and the terrain in Figure 8, which bases on a height-field of 256 x 256 resolution, at 22 fps. The terrain is a rather difficult CSG model because the terrain primitive is not convex and must be rendered with up to eight depth layers.

9 Other Applications

CSG modeling is common practice in offline-rendering systems, whereas interactive modeling environments for such rendering systems have not supported real-time CSG rendering traditionally. With OpenCSG, this is hopefully about to change. Ayam [17], the Tcl/Tk-based modeler for RenderMan compliant renderers, has been the first such application we are aware of that has integrated CSG rendering with OpenCSG in its most recent version (see Figure 6). Currently, the Blender [18] development community discusses integration of an OpenCSG-based rendering plug-in for real-time preview of CSG models.

With the availability of an open and stable CSG library, we foresee many novel uses of CSG. We have implemented some examples based on an OpenCSG plug-in for the scene graph library VRS [19]. One application for CSG can be found in the entertainment field, for example in puzzle games such as the jigsaw puzzle for the letters “C”, “S”, and “G” in Figure 7: A shape that fits exactly through the three letter-shaped holes is easily constructed by intersecting the extruded 3D-models of each letter.

Another new idea for using CSG beyond the scope of traditional modeling applications is road design in 3D terrain models (Figure 8). The red markers, which can be moved to all six directions interactively, define the course of the road. Thereby two tube-formed CSG primitives S_1 and S_2 compose the 3D-model of the road. The upper tube S_1 is subtracted from the terrain primitive T such that the final CSG expression is $(T - S_1) \cup S_2$. In this way, excavations, embankments, and even tunnels can be manipulated interactively and anticipated at an early design stage.

10 Conclusions

We have shown concepts and implementation of OpenCSG, an open graphics library for image-based rendering of CSG models. Our work demonstrates that image-based CSG becomes a mature enabling technology for novel real-time 3D applications. In particular, we foresee applications in the fields of simulation, 3D modeling, CAD/CAM, and gaming.

OpenCSG also shows how a compact abstract API can provide access to a broad range of CSG algorithm variants with a minimum programming effort from an application developer's point-of-view: Due to its simple API, OpenCSG can be easily integrated into any OpenGL-based application. Provided the application-defined 3D shapes are solid, they can be directly processed by OpenCSG as CSG primitives.

The OpenCSG library has been developed since mid-2002 with the first public version released in September 2003 and version 1.0 in May 2004. It is provided under the GPL-license at `http://www.opencsg.org`.

Acknowledgements

We would like to thank Randolf Schultz for providing us with a high-quality screenshot of Ayam.

References

[1] *POV-Ray, The Persistence of Vision Raytracer*, `http://www.povray.org`

[2] S. Upstill, *The Renderman Companion*, Addison Wesley, 1989, ISBN 0201508680

[3] A. A. G. Requicha, *Representations for Rigid Solids: Theory, Methods, and Systems*, ACM Computing Surveys, 12(4):437-464, 1980.

[4] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs, *Near Realtime CSG Rendering Using Tree Normalization and Geometric Pruning*, IEEE Computer Graphics and Applications, 9(3):20-28, 1989.

[5] N. Stewart, G. Leach, and S. John, *Linear-time CSG Rendering of Intersected Convex Objects*, Journal of WSCG, 10(2):437-444, 2002.

[6] T. F. Wiegand, *Interactive Rendering of CSG Models*, Computer Graphics Forum, 15(4):249-261, 1996.

[7] F. Kirsch, and J. Döllner, *Rendering Techniques for Hardware-Accelerated Image-Based CSG*, Journal of WSCG, 12(2):221-228, 2004.

[8] N. Stewart, G. Leach, and S. John, *An Improved Z-Buffer CSG Rendering Algorithm*, 1998 Eurographics / SIGGRAPH Workshop on Graphics Hardware, ACM, 25-30, 1998.

[9] M. J. Kilgard, (editor), *NVIDIA OpenGL Extension Specifications*, April 2004. `http://developer.nvidia.com`

[10] D. Epstein, F. Jansen, and J. Rossignac, *Z-Buffer Rendering from CSG: The Trickle Algorithm*, IBM Research Report RC 15182, 1989.

[11] S. Guha, S. Krishnan, K. Munagala, and S. Venkatasubramanian, *Application of the Two-Sided Depth Test to CSG Rendering*, ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics, 177-180, 2003.

[12] *TGS SolidViz 1.2*, part of TGS OpenInventor 5.0 from Mercury, 2004, `http://www.tgs.com/support/oiv_doc`

[13] M. Segal, K. Akeley, *The OpenGL Graphics System: A Specification*, 2004, `http://www.opengl.org/documentation/spec.html`

[14] *GLEW: OpenGL Extension Wrangler Library*, `http://glew.sourceforge.net`

[15] *RenderTexture 2.0*, `http://gpgpu.sourceforge.net`

[16] C. Wynn, *OpenGL Render-to-Texture*, Presentation, NVidia Corporation, 2002, `http://developer.nvidia.com`

[17] *Ayam*, `http://ayam.sourceforge.net`

[18] *Blender*, `http://www.blender3d.com`

[19] *VRS – The Virtual Rendering System*, `http://www.vrs3d.org`

FreeVGA: Architecture Independent Video Graphics Initialization for LinuxBIOS*

Li-Ta Lo, Gregory R. Watson, Ronald G. Minnich
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM 87545
{`ollie, gwatson, rminnich`}`@lanl.gov`

Abstract

LinuxBIOS is fast becoming a widely accepted alternative to the traditional PC BIOS for cluster computing applications. However, in the process it is gaining attention from developers of Internet appliance, desktop and visualization applications, who also wish to take advantage of the features provided by LinuxBIOS, such as minimizing user interaction, increasing system reliability, and faster boot times. Unlike cluster computing, these applications tend to rely heavily on graphical user interfaces, so it is important that the VGA hardware is correctly initialized early in the boot process in additional to the hardware initialization currently performed by LinuxBIOS. Unfortunately, the open-source nature of LinuxBIOS means that many graphic card vendors are reluctant to expose code relating to the initialization of their hardware in the fear that this might allow competitors access to proprietary chipset information. As a consequence, in many cases the only way to initialize the VGA hardware is to use the vendor provided, proprietary, VGA BIOS. To achieve this it is necessary to provide a compatibility layer that operates between the VGA BIOS and LinuxBIOS in order to simulate the environment that the VGA BIOS assumes is available. In this paper we present our preliminary results on FreeVGA, an x86 emulator based on x86emu that can be used as such a compatibility layer. We will show how we have successfully used FreeVGA to initialize VGA cards from both ATI and Nvidia on a Tyan S2885 platform.

1 Introduction

LinuxBIOS [9] is an open-source replacement for the traditional PC BIOS. The PC BIOS was developed in the 1980's for the original IBM PC, and much of the functionality needed to support this legacy hardware still remains in the PC BIOS today. In addition, the vintage operating systems that ran on these machines were dependent on the BIOS for carrying out many of the configuration activities needed for the system to function properly. Modern operating systems are now able to initialize and configure hardware directly, so there is no longer any reason for the BIOS to be involved. The basic principle behind a modern BIOS, like LinuxBIOS, is to do the minimum necessary to enable system hardware, then leave as much device configuration to the operating system as it can. The result of eliminating this unnecessary initialization is a very fast boot time compared to a traditional BIOS.

Another legacy feature provided by the PC BIOS is a 16-bit callback interface using the x86 software interrupt mechanism. However, only a tiny subset of the interface is used by modern operating systems, and in the case of Linux, it is not used at all. LinuxBIOS does not provide this interface, and as a result, is able to substantially reduce its memory footprint. Compared to 256KB required by the PC BIOS, the typical size of LinuxBIOS is just 32KB to 64KB. This is important because of the small size of FLASH memory in many systems.

Unlike the PC BIOS, only a very small portion of LinuxBIOS is written in assembly code. For the x86 architecture, this is just enough code to initialize the CPU and switch to 32-bit mode. The rest of LinuxBIOS is written in the C language. This makes LinuxBIOS very portable across different architectures, and it has already been ported to support the Alpha and PowerPC processors. Using a high-level language also allows LinuxBIOS to employ a much more sophisticated object oriented device model, similar to the one used in the Linux kernel. In such a model, each physical device has a corresponding software object. The object encapsulates information about the physical device and has methods to probe, initialize and configure the device. The main function of

*Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36, LA-UR N0. 04-7503

LinuxBIOS is really just to organize, query and manage these device objects. This kind of device object model is unheard of in a BIOS implemented in assembly code.

LinuxBIOS has been successfully deployed in a number of real world applications. At Los Alamos National Laboratory, we have Pink and Lightning [11], two very large production cluster systems that use LinuxBIOS. There are also a number of companies shipping commercial LinuxBIOS-based systems. The advantages of LinuxBIOS have also drawn attention from Internet appliance, desktop, and visualization platform developers. These applications have created a demand for video graphics adapter (VGA) card [7] support under LinuxBIOS. In order to use LinuxBIOS, systems running these applications need to be able to initialize a wide variety of VGA cards. However, LinuxBIOS does not have this capability because it does not provide the 16-bit callback interface mentioned earlier.

Traditionally, a VGA card is initialized by software known as the VGA BIOS, which is considered an extension of system BIOS. It is loaded by the system BIOS from an expansion ROM located on the VGA card into a specific address in system memory. Control is then transfered to the VGA BIOS, and it uses the 16-bit callback interface to communicate with the system BIOS. Since LinuxBIOS does not provide this interface, a non-traditional way to initialize the VGA device in a LinuxBIOS environment is required. In order to achieve this, we have developed a system known as FreeVGA. FreeVGA uses an x86 emulator to run the VGA BIOS. By using an emulator, we free the VGA initialization from any architecture dependencies, since the emulator can operate on any type of processor. In addition, the emulator greatly simplifies the implementation of the 16-bit callback interface.

To demonstrate the effectiveness of this technique, we have used FreeVGA to initialize two VGA cards, an Nvidia FX 5600 and an ATI Radeon 9800 Pro, running on a Tyan S2885 mainboard. Both video cards and the mainboard are state-of-the-art, so we can be confident that FreeVGA will work for virtually all mainboard/video card combinations.

The rest of this paper is organized as follows. Section 2 describes previous work on supporting the initialization of VGA cards with VGA BIOS in non-traditional ways. Section 3 presents how we used an x86 emulator to execute the VGA BIOS and the necessary modification to the emulator and LinuxBIOS itself. We also examine some of the issues that need to be dealt with in order to support this technique. Section 4 shows how we found out the issues described in Section 3 and the strategy we used to overcome these problems. Finally, Section 5 is a roadmap of our future development.

2 Related Work

Initializing VGA cards in a non-traditional way (i.e. not using the standard VGA BIOS in the normal manner) is not a new problem, and various other open source projects have addressed it in the past. There are a number of reasons why VGA cards need to be initialized in this manner.

Due to the limitation of the traditional initialization process and legacy VGA hardware, only one VGA device can be initialized in a given system. For systems with multiple VGA cards, only the first one is initialized at boot time, other cards have to be *soft-booted* after the operating system is loaded.

Some VGA hardware is fragile, so that a slight error in programming the registers will put the hardware in a non-functional state. A complete re-initialization is then required to bring the hardware back to normal. Normally, the only way to do such a re-initialization is to re-execute the initialization code in the VGA BIOS. Of course it is always possible to reset the whole system as a last resort, but usually this is not an option. In these cases it must be possible to reset just the VGA device while other parts of the system are still running.

Other open source projects that have addressed the need to initialize VGA cards are described in the following sections.

2.1 SVGALib

SVGALib [2] is a library that provides a generic VGA interface for older VGA cards. It includes a utility called `vga_reset` to re-initialize VGA cards. The utility uses the vm86 mode of x86 processors to execute the VGA BIOS. In vm86 mode, an executing program is just like any other program executing in 32-bit mode, but it executes 16-bit code like a traditional 8086 CPU. To support this, Linux provides a system call to switch a process into vm86 mode. `vga_reset` first maps in the BIOS code and data area from physical memory space to its virtual memory space. Then it sets up register values for instruction and stack pointers. Finally, it enters vm86 mode by calling the vm86 system call.

By default, both VGA BIOS and system BIOS callbacks are executed natively by the hardware, except some privileged instructions and I/O operations. By giving different flags when entering the vm86 mode, it is possible to choose to intercept I/O and BIOS calls. This feature was used frequently in the early stage of the development of our solution as a debugging and verification tool. The I/O and BIOS call logs from `vga_reset` and x86emu were compared to examine if both `vga_reset` and x86emu had the same code execution path in the same hardware environment. If they both had the same execution path, it indicated that the emulator executes the VGA BIOS exactly as the real hardware.

The disadvantage of `vga_reset` is that the vm86 mode is not supported by the AMD x86_64 architecture. The 64-bit Linux kernel does not provide the vm86 system call. During our development, we had to install a 32-bit Linux distribution on our 64-bit AMD K8 platform to run `vga_reset`.

2.2 ADLO

ADLO [5] was the original effort to add VGA BIOS support into LinuxBIOS. ADLO uses the BOCH BIOS to replace the traditional system BIOS. It loads the BOCH BIOS and VGA BIOS into the memory addresses where the traditional system and VGA BIOS are loaded. ADLO then switches the processor back to 16-bit mode and jumps to the entry point of the BOCH BIOS. The BOCH BIOS tries to do the same initialization process and to provide the same BIOS callbacks as a traditional BIOS. The net effect of this BOCH+VGA BIOS combination is like a software reboot in a traditional BIOS system. One of the advantages of ALDO is that it can support legacy operating systems like Window 2000. The problem of ADLO is that it depends on the BOCH BIOS, which is very difficult to maintain and modify. Even adding a message printing statement in the BOCH BIOS will make it fail to build.

2.3 VIA/EPIA Port

Another effort to use VGA BIOS to initialize VGA hardware is the VIA/EPIA port of LinuxBIOS. The port uses a *trampoline* to switch back and forth between the 16-bit and 32-bit modes of x86 processors. This allows the VGA BIOS to be executed directly in 16-bit mode but standard BIOS callbacks to be emulated in 32-bit mode in the C language. Before executing the VGA BIOS, a 16-bit interrupt descriptor table (IDT) is set up to redirect all interrupt calls to the trampoline. The BIOS then switches to 16-bit mode and jumps to the entry point of VGA BIOS. When the VGA BIOS calls the standard BIOS callbacks, the trampoline switches to 32-bit mode, and dispatches the call to the BIOS emulation code. After the emulation code returns, the trampoline switches back to 16-bit mode and returns to VGA BIOS. The main limitation with this approach is that it is highly architecture specific, so can't be used for non-x86 based architectures. The other disadvantage of this method is that because the trampoline is inside LinuxBIOS, it is more difficult to debug than a user space program like x86emu.

2.4 XFree86

The XFree86 project [4] has to solve this problem in order to support multiple-card, multiple-screen configurations. Since XFree86 supports multiple architectures, the solution must be able to initialize VGA hardware not only on x86 systems, but also on other architectures like Alpha and PowerPC. To achieve this, XFree86 uses x86emu emulator to execute the VGA BIOS directly. X86emu is an x86 instruction emulator which does not emulate any hardware other than the core x86 processor in 16-bit mode. The emulator provides helper function stubs to access I/O and memory spaces. XFree86 implements these helper functions in architecture dependent ways. XFree86 first *unmaps* the primary VGA card from physical I/O and memory spaces by programming well known legacy I/O ports or registers in the PCI configuration space. Then it maps in the I/O and memory spaces of the secondary card, loads the VGA BIOS image of the card into the *virtual* memory space of the emulator. The emulator then executes the VGA BIOS starting from the entry point. Because most VGA BIOSes also require traditional BIOS callbacks which are not available in non-x86 systems and not usable in 32-bit x86 systems, the emulator also has to intercept these BIOS calls and then emulate them in a similar way as I/O and memory accesses.

In a summary, each of these previous efforts was found to have deficiencies. Both SVGALib and VIA/EPIA are non-portable, so are not suitable for integration in LinuxBIOS. SVGALib, ADLO and VIA/EPIA were found to be very difficult to debug, which is a major problem for a complex system like LinuxBIOS. XFree86 initializes the VGA hardware very late, which does not meet our design goals. These problems motivated us to seek a solution which was able to address all these issues, while taking advantage of the experience gained by the previous research.

The concept and advantages of using an emulator to execute the VGA BIOS in order to initialize VGA hardware are obvious. The solution is portable across different platforms, and it is flexible enough to monitor I/O and memory accesses and BIOS callbacks. The ability to monitor these accesses is very useful for debugging purposes. As a consequence, the solution we have chosen for FreeVGA is based on a modified version of x86emu.

3 VGA Emulation

Most modern VGA cards support two modes of operation: legacy mode and native mode. In legacy mode the card replicates the graphics hardware interface that was used on the original IBM PC/AT. The legacy mode is primarily used to provide a compatibility mode so that applications and drivers have a common programming interface. In native mode, the card provides access to a vendor specific register interface that is used to configure and control the card. Most VGA cards require an elaborate programming sequence to initialize the VGA hardware and turn it to legacy mode. If LinuxBIOS was to support direct initialization of the card, it would need to use a device driver that provided this programming sequence. Unfortunately, most vendors worry that even exposing the interface to their proprietary hardware will allow their competitors to plunder their intellectual property; hence they do not reveal the sequence of register diddles necessary to initialize their cards.

Linux uses a frame buffer device as an abstraction of graphics hardware. Applications interact with frame buffer device interface (`/dev/fb`) instead of directly accessing the hardware. At a minimum, the frame buffer device driver provides support to switch video modes and some basic drawing functionality. Some vendors like Matrox provide a *sophisticated* frame buffer device driver which can initialize the hardware from the power-up state to any video mode available by the hardware. Our original intention was to persuade vendors to provide these sophisticated drivers so that they could be used by LinuxBIOS. However many of the vendors who provide enough information to implement a minimal frame buffer device driver hesitate to provide the additional information necessary for a such a driver.

As a result, the only way to reliably initialize the hardware from power-up in a vendor-neutral manner is to run the vendor supplied VGA BIOS. Once the VGA BIOS has been run, the card will switch to legacy mode and it can be controlled using the legacy interface from then on. Depending on the implementation of the VGA BIOS, the 16-bit BIOS callback interface may be used to communicate with the system BIOS. Since LinuxBIOS lacks this callback interface, it can not support VGA BIOS directly in the same way as the traditional system BIOS. We have to either add the 16-bit callback interface to LinuxBIOS or use another software to provide this interface. The use of an emulator solves this problem by allowing the emulator to run in 32-bit mode to execute the 16-bit mode VGA BIOS and then implement the callback interfaces as necessary.

3.1 x86emu

The emulator we used to enable VGA support in LinuxBIOS was based on a modified version of x86emu. The x86emu emulator was originally developed by SciTech Software [1] as part of their SciTech SNAP SDK. The XFree86 project adopted the emulator for soft-booting VGA cards in their X-server. We used the XFree86's version rather than the SciTech's version because it is more updated and debugged.

The virtual machine of the emulator is implemented by a data structure representing all the integer and floating point registers in an x86 CPU. The emulator decodes and jumps to an entry of a function table based on the first op code of each instruction. The functions in the function table update the virtual machine with the outcome of the *execution* of the instruction. The emulator uses helper functions provided by client applications to communicate to the real world, for instance, accessing I/O and memory spaces. The emulator allows interrupt handling using either an interrupt handler provided by the client application or an interrupt handler in the BIOS.

IO and Memory Access The client application provides a set of functions for accessing I/O ports and another set of functions for accessing memory addresses. The structures used to define the functions are shown below:

```
typedef struct {
    u8    (inb)(int addr);
    u16   (inw)(int addr);
    u32   (inl)(int addr);
    void  (outb)(int addr, u8 val);
    void  (outw)(int addr, u16 val);
    void  (outl)(int addr, u32 val);
} X86EMU_pioFuncs;
```

```
typedef struct {
    u8    (rdb)(u32 addr);
    u16   (rdw)(u32 addr);
    u32   (rdl)(u32 addr);
    void  (wrb)(u32 addr, u8 val);
    void  (wrw)(u32 addr, u16 val);
    void  (wrl)(u32 addr, u32 val);
} X86EMU_memFuncs;
```

These two sets of functions are installed into the emulator via `X86EMU_setupPioFuncs()` and `X86EMU_setupMemFuncs()` respectively.

Since we are working on an x86 platform, the implementation of the I/O access functions is just a thin wrapper for the inline assembly functions provided in `sys/io.h`. All I/O operations are directed to the physical I/O ports without any intervention or emulation.

In our setup, we statically allocated 1MB of memory to be used as the virtual memory of the emulator. The memory access functions direct all memory accesses made by VGA BIOS to this area, except accesses to the legacy VGA buffer. These are directed to another virtual memory area which is mmaped from `/dev/mem`. We implemented the memory access functions by reading and writing these two memory regions according to an address passed as argument to these access functions.

Interrupt Handling In the x86 architecture, there are 256 software interrupts. The client application provides an array of 256 functions to the emulator for interrupt handling, in a similar way as I/O and memory access functions. When the emulator encounters an `INT` instruction with an interrupt number `N`, it calls the `N`th entry of the array. The interrupt handling function can choose to handle the interrupt by itself or let the emulator execute the handler in its virtual memory.

In our implementation, all software interrupts are directed to a single `do_int()` function. When this function is called with a interrupt number, it first checks if there is any handler installed by the VGA BIOS for that interrupt number. If there is no handler installed, it will call the default emulation code implemented in the C language, otherwise it will execute the handler installed by VGA BIOS with the emulator.

3.2 Legacy VGA Issues

Using x86emu provides LinuxBIOS with a means of initializing the VGA hardware and then switching the card to legacy mode. However there are a number of other issues that need to be addressed when the card is operating in this mode. Figure 1 shows the system and card memory layout when operating in legacy mode.

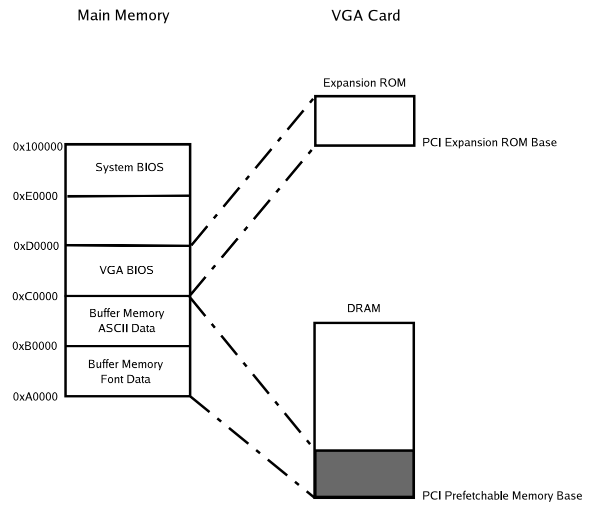


Figure 1: Legacy VGA Memory Map

Buffer Memory The memory on the VGA card is mapped to system physical address space by the prefetchable memory resource in the PCI Configuration Space of the card. The buffer memory used in legacy mode is just a small portion of the whole memory installed on the card, as shown in the shaded area in Figure 1. This portion of memory is mapped to system memory addresses 0xA0000 to 0xBFFFF respectively. Part of this area is used to store bitmap data which is interpreted by the hardware as font information. The rest is used to store the ASCII codes and color values to be displayed on the screen. The VGA BIOS clears and updates the buffer memory during initialization, so it is necessary to map this region of physical memory to the virtual memory space of the emulator.

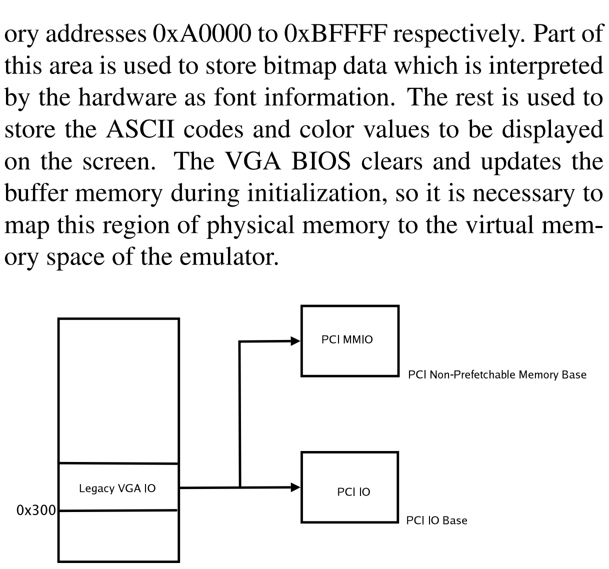


Figure 2: Legacy VGA I/O Map

I/O Addresses The control registers of the VGA device are located by the PCI I/O resource map to the I/O space of the processor, or by the PCI non-prefetchable memory (MMIO) resource map to a range of physical memory addresses. VGA cards also provide access to control registers via the legacy VGA I/O addresses in the range 0x300 to 0x400. Generally, the VGA BIOS will access control registers using both memory mapped I/O and via the legacy VGA I/O ports. To support this, it is necessary to ensure that both I/O access methods are forwarded correctly to the VGA device.

Expansion ROM Before the VGA BIOS can be executed, it has to be loaded from the expansion ROM on the VGA card into the VGA BIOS memory area in system memory. The PCI specification [10] defines the format for the VGA BIOS image and a procedure to load the image as follows:

1. The image in the expansion ROM starts with a 0x55, 0xAA signature.
2. The system BIOS should search the for signature in the expansion ROM and load the image into memory.
3. If the device is a VGA device, the system BIOS is required to load the image to 0xC0000 which is the VGA BIOS area.
4. After loading the image, the system BIOS should jump to the entry point of the image which is at offset 0x3.

The expansion ROM loading was implemented in LinuxBIOS as the initialization methods of PCI devices. LinuxBIOS probes and allocates the expansion ROM resources required by PCI devices in one of the stages of device enumeration. In a later stage, it loads the expansion ROM image from ROM to RAM and uses the emulator to execute the image.

CPU Cache The system memory region allocated to VGA buffer memory (0xA0000-0xBFFFF) is actually aliased in legacy mode. This means that memory accesses in this range can be forwarded to system memory or VGA card memory depending on the setting of the system chipset or the cache controller in the processor.

The cache in x86 processors after Pentium Pro is configured by Model Specific Register (MSR) called Memory Type Range Register (MTRR) in the processor. MTRR controls the cache mechanism of a range of physical address. Addresses under 1MB are controlled by *fixed* MTRRs and addresses above 1MB are controlled by *variable* MTRRs. The fixed MTRRs on the K8 have 2 extension bits (RdMEM and WrMEM) [6] which control the forwarding of read and write access to memory address under 1MB. When set and enabled, the RdMEM bit in the MTRR will forward read access in the range to system memory. The WrMEM bit will forward write access to system memory. However, since we want memory access to the VGA buffer memory be forwarded to the VGA card, we have to clear these two bits in the MTRRs controlling this memory range.

HyperTransport Routing The AMD K8 processor and its chipsets are interconnected by HyperTransport Technology [8]. Each processor has a *northbridge* integrated in the same package. The northbridge connects the core processor to system memory and other parts of the system (via a *southbridge*). On the K8, each northbridge provides three HyperTransport links that can be used for communication. The way the processors and chipsets are connected via these links is determined by mainboard designers and varies from board to board. There is also a HyperTransport routing table in the northbridge which controls how memory and I/O requests are routed. The BIOS has to configure the routing table based on both the physical layout of the HyperTransport hierarchy, and the PCI devices attached to the hierarchy. I/O and memory transactions can then be forwarded to the correct HyperTransport link on the northbridge. For VGA devices, accesses to both the I/O and memory resources defined in the PCI configuration space, and to the legacy VGA, also have to be forwarded correctly.

Figure 3 shows the HyperTransport hierarchy of the Tyan S2885. The mainboard designers have connected the two CPUs together by *Link 1* on each CPU. The

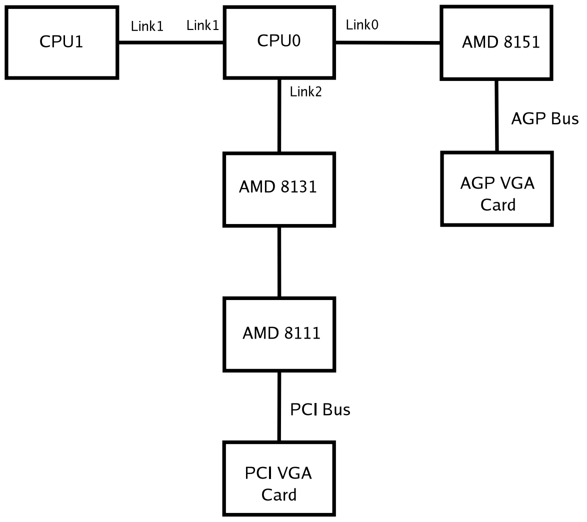


Figure 3: S2885 HyperTransport Hierarchy

AMD 8151 AGP bridge is connected to *Link 0* and the AMD 8131 PCI-X bridge is connected to *Link 2* on CPU 0. *Link 0* and *Link 2* of CPU 1 are left unconnected. LinuxBIOS must configure the routing table based on which kind of VGA card is installed. For example, for a VGA card connected to the AGP, it is necessary to set up the routing table in CPU 0 to forward legacy VGA I/O and memory transactions to *Link 0*. However if the VGA card is connected to the PCI bus, the routing table will need to forward transactions to *Link 2*.

PCI and AGP Bridges PCI and AGP bridges forward I/O and memory transaction from its primary bus to its secondary bus. The range of access to be forwarded is configurable and should cover the PCI I/O and memory resources used by all devices on the secondary bus. Usually this range does not include the I/O amd memory addresses used by legacy VGA. We have to enable forwarding of legacy VGA access in additional to normal PCI access by programming a bit in the Configuration Space of the bridge.

3.3 Integration

In order to achieve our objective of early VGA initialization during the BIOS boot phase, it is necessary to make the emulator part of LinuxBIOS.

User Space Versus Kernel Space One problem we expected to encounter when moving the emulator into LinuxBIOS was operating system and library dependency issues. The emulator was a user space program that used operating system and library calls for memory management, message printing and accessing PCI configuration space. Those functions are not available during the boot phase, and need to be replaced by corresponding supporting routines in LinuxBIOS.

Fortunately, it turned out that the integration process was relatively easy, since there were only a few operating system and standard library dependencies in the emulator itself. Most of the effort was spent on implementing expansion ROM loading in LinuxBIOS, and fixing bugs in I/O and memory transaction forwarding.

Device Object Model The integration fully used the device object model available in LinuxBIOS. This was done in such a way that not only VGA BIOS could be executed, but the same technology could also be used to initialize other devices requiring a proprietary BIOS, for instance, some SCSI controllers.

The emulator was treated as one of the initialization methods of PCI devices. This fitted the emulator nicely into the LinuxBIOS device driver model because the initialization method is automatically called at certain stages of the device enumeration. Once the device enumeration code found a PCI device with expansion ROM, it would call the emulator at the appropriate stage to initialize the hardware. The hardware would then be initialized in the same fashion as any other devices are initialized.

With this technique, the VGA hardware is fully configured before any bootloader payloads are loaded. This means that bootloaders can now use the legacy mode of the VGA device as a console and the Linux console driver works in exactly the same way as in a traditional PC BIOS environment.

Size Overhead Integrating FreeVGA into LinuxBIOS had virtually no impact on the size of the resulting ROM image. The compressed ROM image only increased by 16KB, but because the final ROM image is padded to the nearest power of 2, this increase was absorbed into the existing unused space. The runtime size of the uncompressed image was only increased by 40KB.

4 Testing

Testing of FreeVGA was carried out on a Tyan S2885 mainboard. This mainboard was chosen because it was a state-of-the-art board that uses 64 bit AMD CPUs. It was also the only AMD K8 mainboard with an AMD 8151 AGP bridge and an AGP slot. The AGP slot on the mainboard allowed us to test a range of newer generation AGP VGA cards. The mainboard was configured with dual 1.6 GHz AMD K8 processors. There was 3GB of DRAM installed, 2 GB of the memory was installed on DRAM DIMM connected to CPU 0 and the other 1GB was connected to CPU 1.

One of the challenges of the Tyan was that it was a very complex platform to work with. The first time we tried running the emulator we received nothing at all on the screen. Checking the execution log from the emulator, we found that I/O accesses to the PCI I/O resource region of the VGA device returned meaningful values, but that I/O accesses to the legacy VGA I/O ports on the card always returned invalid values. From this we were able to deduce that the northbridge and AGP bridge were forwarding normal PCI I/O accesses to the card correctly, but accesses to the legacy VGA were not. To test this, we temporarily configured legacy VGA forwarding in the AGP bridge and HyperTransport routing in the northbridge. Re-running FreeVGA at this point resulted in scrambled text on the screen. This was promising, and it verified that HyperTransport routing was going to be crucial for this platform.

Next we tried to alter the scrambled pattern by writing to the buffer memory via both the legacy VGA buffer area and the PCI memory resource region. We found that changes made via the PCI memory resource region were not reflected in the legacy VGA buffer area and vice versa. This was strange because in theory no matter which way the buffer memory was modified, it should updated the same memory on the VGA card. We finally realized that the cache controller in the AMD K8 processor was forwarding the access to the memory on the mainboard rather than on the VGA card. This problem was solved by changing the MTRRs with the help of the kernel MSR driver.

At this point, the screen remained scrambled even though we were sure that the CPU was forwarding accesses correctly. By comparing the contents of the VGA buffer memory when the system was booted with both a tradition BIOS and with LinuxBIOS, we found that the contents of the font data memory were different. This was because the emulator was executing the VGA BIOS in its own virtual memory address, whereas the VGA BIOS tried to update the font data at a physical address. This was solved by modifying the emulator to map the physical memory device `/dev/mem` to its virtual memory address for the legacy VGA buffer.

4.1 Nvidia FX 5600

The first VGA card we tried was an Nvidia FX 5600. This is a high performance 3D graphics card with 256MB of onboard memory.

In additional to the fact that Nvidia is the market leader in VGA chipset design, we choose this card because Nvidia have never publicly released any programming documentation on the hardware. This meant that FreeVGA would not be able to do any direct card configuration and would have to rely totally on running the vendor supplied VGA BIOS. Much to our surprise, executing the VGA BIOS work successfully on the first attempt and we were greeted with the Nvidia banner messages on the screen. It turned out that the VGA BIOS did not do anything unusual, nor did it require any BIOS callbacks. After the successful initialization, we were able to run Linux VGA console without problem.

In order to fully exercise the Nvidia card, we ran a benchmark using the Unreal game [3] under the XFree86 X-window system. This also allowed us to test two different versions of the X server driver, one provided by the XFree86 Project and the one provided by Nvidia. Both drivers worked flawlessly and there was no significant difference running the benchmark as compared to the system booted with a traditional BIOS. This benchmark also exercised the 3D and AGP operation of the card and no problems were found.

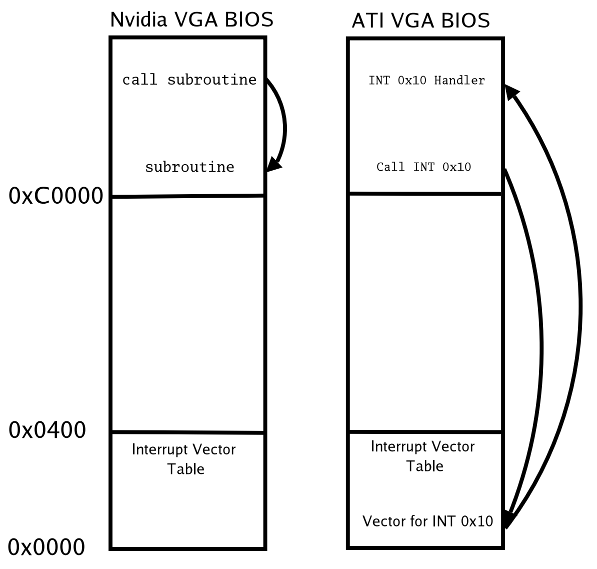


Figure 4: Direct and Indirect BIOS call

4.2 ATI Radeon 9800 Pro

The ATI Redeon 9800 Pro is a high performance 3D graphics accelerator card with 256MB of memory. The card was manufactured and supplied by Tyan. It is comparable in performance to the Nvidia card.

The first time we tested the emulator on the ATI card it crashed the system. By examining the I/O logs we found that during the initialization, the VGA BIOS was setting the video mode and then displaying some messages on the screen. As shown in Figure 4, the Nvidia BIOS did this by calling subroutines in the BIOS directly. In the case of the ATI card, the VGA BIOS tried to install a BIOS callback handler in the interrupt vector table and then call the handler using the software interrupt mechanism. Although using a BIOS callback was not a problem for FreeVGA, the problem was that we were intercepting the software interrupt and implementing our own handler base on our limited knowledge of legacy VGA. Obviously, the way we implemented the handler was incorrect for the ATI hardware and caused the crash. Once we stopped intercepting the BIOS call and executed the handler in the ATI BIOS, the card was initialized without problem.

At this point we ran the same benchmark on the ATI card. This verified that there were no differences in the performance and operation of the card compared to a system booted with the traditional BIOS.

5 Future Work

The work to date has shown that it is possible to use our methodology to reliably initialize two very different VGA cards. Because of the different nature of the cards, and the fact that we have not needed any vendor input to achieve this result, we are confident that this technique will apply to virtually any type of VGA card. However there are still a number of issues that need to be addressed before FreeVGA is ready for general use.

Chipset Dependencies At the time of writing, we have only tested the emulator on an AMD K8 platform. Each vendor chipset has its own, very different, way of caching the frame buffer memory and forwarding I/O and memory accesses. The HyperTransport architecture of the AMD K8 is the most complicated one we have ever seen so far, however there is no guarantee that the techniques we have used here will be applicable to other chipsets. More testing is required so that that the experience we have gained on AMD K8 can be extended to the support of other chipsets.

Other Architectures One of the advantage of FreeVGA is its architecture independence. Since LinuxBIOS already supports other non-x86 architectures, such as the PowerPC, it will be necessary to port FreeVGA support these other architectures too. Currently, VGA cards have to be programmed using OpenFirmware instead of the x86 VGA BIOS in the expansion ROM to be usable on PowerPC. This has severely limited the choice of VGA cards on PowerPC-based system. By using FreeVGA to initialize VGA cards on PowerPC, the same VGA cards that are available for the x86 architecture will be available for the PowerPC.

The main issue of porting the emulator to these architectures is that they have very different ways of accessing legacy VGA IO and memory. The legacy VGA IO and memory are mapped to physical memory address in a chipset and mainboard dependent way. The mechanism of accessing PCI Configuration Space is different from x86 architecture too. We expect a much more complicated implementation of `X86EMU_pioFuncs` and `X86EMU_memFuncs` for these architectures.

6 Conclusion

In this paper, we have described FreeVGA, an architecture independent method for initializing video graphics adapter cards. The technique was developed so that LinuxBIOS, an open source replacement for the traditional PC BIOS, would be able to initialize graphics hardware very early in the boot process. To achieve this, FreeVGA uses an x86 emulator based on x86emu to run the actual VGA BIOS from the graphics card. This ensures that the card is initialized correctly, and does not require any knowledge of proprietary hardware information.

FreeVGA has been successfully tested using a Tyan S2885 mainboard configured with both ATI Radoen 9800 and Nvidia FX 5600 cards. Our testing showed that these cards could be successfully initialized with FreeVGA, and then support the operation of the XFree86 X-window system without any problems. Both the AGP and 3D features of the cards were completely operational, and benchmarking showed no performance difference compared to the system booted with the standard PC BIOS. Although there are still a number of issues to be addressed to enable seamless integration with LinuxBIOS, the results of our testing gives us great confidence that FreeVGA will be an effective, vendor independent, alternative for initializing VGA hardware on a range of different platforms.

7 Acknowledgment

The author would like to thank David Hedricks for setting up the development platform and performing the elaborate testing.

Tyan Computer Corp. kindly provided the S2885 mainboard and the ATI Radeon 9800 Pro card. Tyan is a long time supporter of the LinuxBIOS project, and ships LinuxBIOS on a number of its mainboard products.

References

[1] http://www.scitechsoft.com/.

[2] http://www.svgalib.org/.

[3] http://www.unrealtournament.com/.

[4] http://www.xfree86.org.

[5] Adam Agnew, Adam Sulmicki, Ronald Minnich, and Willian Arbaugh. Flexibility in rom: A stackable open source bios. In *2003 USENIX Annual Techinical Conference*, San Antonio, Texas, USA, June 2003.

[6] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, May 2003.

[7] Richard F. Ferraro. *Programmer's Guide to the EGA, VGA, and Super VGA Cards*. Addison Wesley, 1994.

[8] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification*, January 2003.

[9] Ron Minnich, James Hendricks, and Dale Webster. The Linux BIOS. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[10] PCI-SIG. *PCI Local Bus Specification*, December 1998.

[11] Gregory R. Watson, Matthew J. Sottile, Ronald G. Minnich, Erik A. Hendriks, and Sung-Eun Choi. Pink: A 1024-node single-system image linux cluster. In *Proceedings of HPC Asia 2004*, Toyko, Japan, July 2004.

The Ethernet Speaker System [*]

David Michael Turner and Vassilis Prevelakis

Computer Science Department
Drexel University
`{dmt36, vp}@drexel.edu`

Abstract

If we wish to distribute audio in a large room, building, or even a campus, we need multiple speakers. These speakers must be jointly managed and synchronized. The Ethernet Speaker (ES) system presented in this paper can be thought of as a distributed audio amplifier and speakers, it does not "play" any particular format, but rather relies on off-the-shelf audio applications (*e.g.,* mpg123 player, Real Audio player) to act as the audio source. The Ethernet Speaker, consists of three elements: (a) a system that converts the audio output of the unmodified audio application to a network stream containing configuration and timing information (rebroadcaster), (b) the devices that generate sound from the audio stream (Ethernet Speakers), and (c) the protocol that ensures that all the speakers in a LAN play the same sounds.

This paper covers all three elements, discussing design considerations, experiences from the prototype implementations, and our plans for extending the system to provide additional features such as automatic volume control, local user interfaces, and security.

Keywords: virtual device drivers, OpenBSD, audio, multicast.

1 Introduction

Consider a situation where you want to listen to some audio source in various rooms in your house, alternatively you may want to send audio throughout a building. In such situations the traditional approach would be to set up amplifiers, speakers and connect them all up into one large analog audio network (*e.g.,* [4]). If laying wires is not an option, then wireless solutions also exist where the audio signal is sent over radio frequencies and the speakers are essentially radios that listen on a preset frequency.

In this paper we discuss our implementation of a similar architecture using an Ethernet network and small embedded computers as speakers (Figure 1).

Our motivation for undergoing this project was that we believe that using an existing network infrastructure may allow the deployment of large scale public address systems at low cost. Moreover, having an embedded computer next to each speaker offers numerous opportunities for control of the audio output, such as remote playback channel selection, volume levels, perhaps even the ability to use the built-in microphone to determine the appropriate volume level depending on ambient noise levels.

Numerous solutions exist for transferring audio over the Internet, but since most local networks are Ethernet-based we wanted to use this experiment to determine whether we can come up with a number of simplifications to the architecture by assuming that all the speakers are located on the same Ethernet segment. This ties

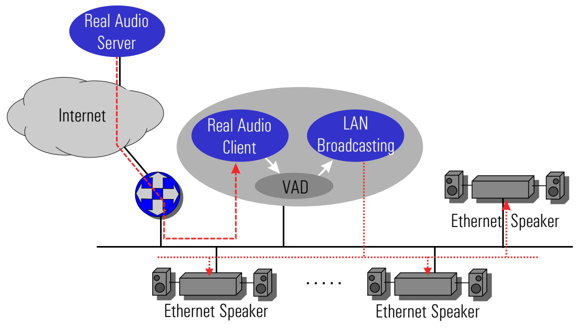


Figure 1: Rebroadcasting WAN Audio into the LAN.

[*]This work was supported by NSF under Contract ANI-0133537.

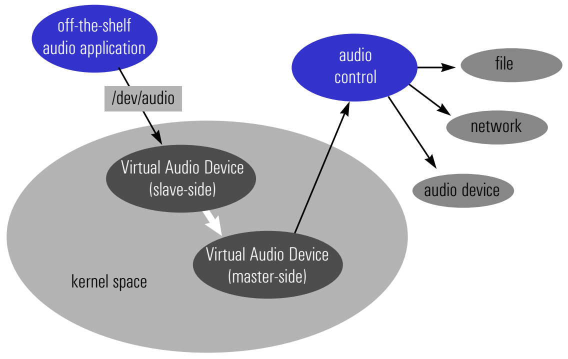

off-the-shelf audio application
/dev/audio
audio control
file
network
audio device
Virtual Audio Device (slave-side)
Virtual Audio Device (master-side)
kernel space

Figure 2: Overview of the Virtual Audio Device.

in very well with the greatly improved range and size of modern Ethernet LANs. In section 2 we discuss the impact of this assumption on the design and implementation of our system.

Another issue that we faced was how to generate the audio streams that would feed these (Ethernet) speakers. We did not want to design yet another audio library or streaming service, or to have to modify existing applications so that they would use our system. The solution we selected was to redirect the audio stream from inside the kernel so that instead of going out the built-in audio card, the stream would be channelled directly to the network or to some other user-level application. This led to the design of the virtual audio device (VAD) that virtualizes the audio hardware in the same way as the Unix system uses pseudo terminals (`pty(4)`).

The advantage of this approach is that the application cannot determine whether it is sending the audio to a physical device or to a virtual device and hence off-the-shelf applications (even proprietary ones) can be used to send audio to our Ethernet Speakers (Figure 2).

In the rest of this paper we describe the basic elements of the Ethernet Speaker (ES) system, including the VAD, the communications protocol used to send audio data and configuration information to the Ethernet Speakers and the platform used to implement the ES. We also describe our prototype and the lessons we learned from its implementation. Finally, we discuss some of our plans for adding some security and remote management features to the system.

2 Design

2.1 The Virtual Audio Device (VAD)

In this section we discuss the modifications we made to the OpenBSD kernel to support the virtual audio device (VAD). Although we have used the OpenBSD kernel, our modifications should be easy to port to any system that uses a similar architecture for the audio driver.

The audio subsystem of most modern Operating Systems provides a path from the application to the audio hardware. While this arrangement works most of the time, it has a number of deficiencies that can be quite frustrating. These include:

- The numerous encoding formats of the audio data require the use of different audio players, each with its own user interface. Some of the players (*e.g.,* real audio player) offer only graphical user interfaces making them unusable on embedded platforms or systems that offer only character-based user interfaces.
- The tight binding between the audio device and the audio hardware means that the sound must be generated close to the computer running the audio application.
- Certain streaming services offer no means of storing the audio stream for later playback (time shifting).

Despite the numerous audio formats in use, the services offered by the audio device driver (`audio(4)`) are well defined and relatively straightforward in terms of formats and capabilities. This implies that by intercepting an audio stream at the kernel interface (system call level), we only need to deal with a small set of formats. In other words, the various audio applications perform the conversion from the external (possibly proprietary) format to one that is standardized. We, therefore, require a mechanism that allows the audio output of a process to be redirected to another process. The redirection occurs inside the kernel and is totally transparent to the process that generates the audio stream. Our system utilizes a virtual audio device (VAD) that plays a role similar to that of the pseudo terminals that may be found on most Unix or Unix-like systems. The VAD consists of two devices (a master and a slave). The slave device looks like a normal audio device with the difference being that there is no actual sound hardware associated with it.

The audio application opens the slave device and uses `ioctl` calls to configure the device and `write` calls to send audio data to the device. Another application can then open the master device and read the data written to the slave part (Figure 2).

2.1.1 The OpenBSD Audio System

The OpenBSD audio driver is an adaptation of the NetBSD audio driver and consists of two parts: the device independent high level driver and the device dependent low level driver. The high level driver deals with general issues associated with audio I/O (*e.g.,* handling the communications with user-level processes, inserting silence if the internal ring-buffer runs out of data, *etc.*), while the low-level drives the audio hardware. User-level applications deal entirely with the high-level audio driver.

In the OpenBSD kernel there is one instance of the high-level audio driver and as many instances of the low-level as types of audio cards connected to the system. The first audio device is associated with the `/dev/audio0` device, the second with `/dev/audio1` and so on.

Applications use `ioctl` calls to set various parameters (such as the encoding used, the bit rate, *etc.*) in the driver and the usual file I/O calls to read and write data to the device.

The interface between the two levels of the audio device driver is well documented (`audio(9)`) so adding a new audio device is fairly straightforward.

In the case of the Virtual Audio Drive, we had to construct a low-level audio device that is recognised by the OpenBSD kernel as a valid device. Using terminology borrowed from the pseudo terminal implementation (`pty(4)`), we note that the VAD driver provides support for a device-pair termed a virtual audio device. A virtual audio device is a pair of audio devices, a master device and a slave device. The slave device provides to a process an interface identical to that described in `audio(4)`. However, whereas all other devices which provide the interface described in `audio(4)` have a hardware device of some sort behind them, the slave device has, instead, another process manipulating it through the master half of the VAD. That is, anything written on the slave device (`vads`) is given to the master device (`vadm`) as input (currently `vads` only supports audio output).

The master (or control) device has its own entry in the `/dev` directory providing an access point for user-level applications.

In addition to the audio data, the slave device passes control information (*e.g.,* values set using the `ioctl(2)` system call) to the control side. Thus the application accessing `vadm` can always decode the audio stream correctly.

2.1.2 Why a Virtual Audio Device?

An audio application does more than just send audio data to the audio device. The control information, mentioned above, is vital to the correct playback of the audio stream. The audio application establishes these settings by configuring the audio device. Moreover the application may check the status of the audio device from time to time. All of the above indicate that we cannot simply replace the audio device (`/dev/audio`) with something like a named pipe; our redirector must behave like a real audio device and be able to communicate the configuration updates to the process that receives the audio stream via the master side of the VAD.

2.2 The Audio Stream Rebroadcaster

The data extracted from the master side of the VAD may be stored in a file, processed in some way and then sent to the physical audio device, or transmitted over the network to the Ethernet Speakers. In this section we discuss the Audio Stream Rebroadcaster which is an application that gateways audio information received from the public Internet to the local area network (see Figure 3).

In addition to providing input to the Ethernet Speakers, the rebroadcaster service may be used as a proxy if the hosts in the LAN do not have a direct connection to the public Internet and thus require the use of a gateway (which has access to both the internal and the external networks) to rebroadcast the audio stream.

Another reason may be that if we have large numbers of internal machines listening to the same broadcast, we may not want to load our WAN link with multiple unicast connections from machines downloading the same data. By contrast, the rebroadcaster can multicast the data received from a single connection on the WAN link.

Moreover, by having a known format for the multicast

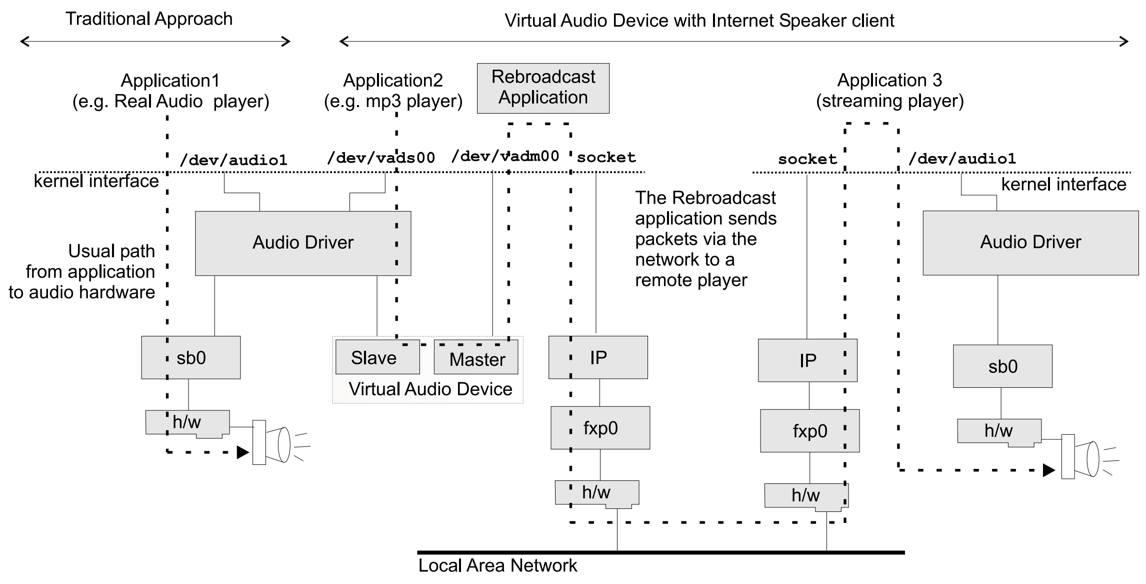


Figure 3: Application 1 plays audio to the normal audio device, while Application 2 plays to the VAD, which sends the audio data to a remote machine via the network.

audio data, we can play back any stream with a single player. These players do not need to be updated when new audio formats are introduced.

The Audio Steam Rebroadcaster consists of two applications, the producer that runs on the same machine as the virtual audio device (and hence the application that receives the audio stream over the public Internet) and one or more consumers running on other hosts in the LAN. The producer sends the audio stream as multicast packets to reduce the load on the network.

Early versions of our design [10], sent onto the network the raw data as it was extracted from the VAD. However this created significant network overhead (around 1.3Mbps for CD-quality audio). On a fast Ethernet this was not a problem, but on legacy 10Mbps or wireless links, the overhead was unacceptable. We, therefore, decided to compress the audio stream.

Fortunately, there are a wide variety of audio compression technologies that more than satisfy our needs. We decided on the use of Ogg Vorbis (w.xiph.org) for multiple reasons. It is completely patent free and it's implementation is distributed under the General Public License. Additionally, being a lossy psycho-acoustic algorithm, it provides excellent compression, comparable to proprietary solutions such as MP3.

The introduction of Ogg Vorbis into the Ethernet speaker architecture brings about several interesting considerations. It is commonly known that the combination of multiple lossy codecs onto the same set of data can lead to greater quality loss than necessary. This is because the algorithms may choose to eliminate very different segments of data in order to achieve the same goal. The best one can hope for would be that the audio quality would not get any worse. If a user were to take their favorite MP3 file and play it over the Ogg Vorbis equipped Ethernet Speaker it would pass through two very different lossy audio compression algorithms. In order to try and compensate for this loss of quality we simply set the Ogg Vorbis quality index to its maximum. This causes the algorithm to throw away as little data as possible while still providing adequate compression. Luckily, our experience so far has not revealed any audible defects to the stream.

Audio channels with low bit-rates are still sent uncompressed because the use of Ogg Vorbis introduces latency and increases the workload on the sender (Figure 4). The selective use of of compression can be enhanced by allowing the rebroadcast application to select the Ogg Vorbis compression rate. This will allow more aggressive compression to be performed on high bit-rate audio channels where audio quality is less of a concern.

2.3 The Communication Protocol

Early in the design of the Ethernet Speaker System, we decided that the communication will be restricted to a single Ethernet LAN. The reason for this decision was that a LAN is a much friendlier environment to communicate providing low error rates, ample bandwidth, and most importantly, well behaved packet arrival. So far at Drexel, even though the main campus network often sees large peaks in traffic, we have not experienced packet loss or transient network disruptions that allowed the input buffer of the ESs to empty and thus affect the audio signal.

Another advantage of using a LAN is that we get multicast support by default. Once packets have to cross routers, then multicasting becomes an option few network administrators are likely to support. Nevertheless, by using multicast packets, we avoid the need to have the Ethernet Speakers contact the Rebroadcaster in order to receive the audio stream. As noted earlier, the VAD is sending configuration information as well as audio data. The configuration information must be passed on to the Ethernet Speakers so that they can decode the audio stream. We do this by having the Rebroadcaster send control packets at regular intervals with the configuration of the audio driver. The Ethernet Speaker has to wait till it receives a control packet before it can start playing the audio stream.

The advantage of this approach is that (a) the Rebroadcaster does not need to maintain any state for the Ethernet Speakers that listen in, and (b) clients do not need to contact the Rebroadcaster to retrieve the audio configuration block.

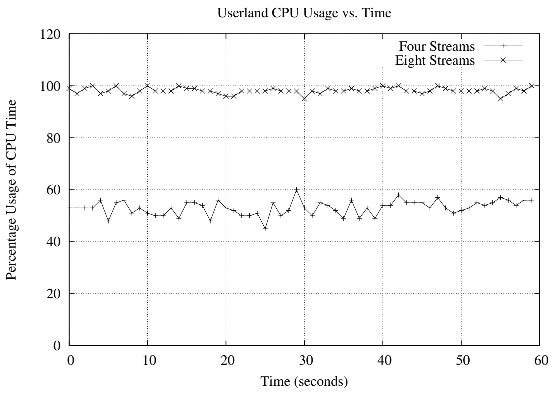


Figure 4: Compression impact on CPU load, as we increase the number of compressed streams transmitted by the local rebroadcaster. Each stream is a separate CD-quality stereo audio stream.

In this way our Ethernet Speakers function like radios, *i.e.,* receive-only devices. This design requirement simplifies the architecture of the audio producer enormously: the Rebroadcaster is just a single-threaded process that collects audio from the master-side VAD and delivers it to the LAN.

2.4 The Ethernet Speaker

The task of refining the design of the ES runtime environment was made much easier by the fact that we had previous experience with such systems from work in embedded systems such as VPN gateways [11] and network monitoring stations [12]

We were, thus, aware that the ES has to be essentially maintenance-free so that once deployed, the administrators will not have to deal with it. This requirement in turn leads to two possible configurations: one that boots off the network or one that boots off a non-volatile RAM chip (Flash memory).

In either case the machine will have to receive its configuration from the network. Network setup may be done via DHCP, but we also need additional data such as the multicast addresses used for the audio channels, channel selection, etc.

We established that the machine runtime should be based on a ramdisk configuration. The rationale behind this decision is that if we use a Flash boot medium, we would not be able to have it mounted read-write because a power (or any other) failure may create a non-bootable machine. In the case of the network-based system, having machines mount their root and swap filesystems over the network would lead to scalability problems.

The requirement that we should be able to update the software on these machines without having to visit each machine separately made the network boot option more appealing.

We thus decided to use a ramdisk-based kernel that is loaded over the network. The ramdisk is part of the kernel, so that when an ES loads its kernel, it gets the root filesystem and a set of utilities which include the rebroadcast software. The ramdisk contains only programs and data that are common to all ESs. Each machine's network-related configuration is acquired via

DHCP, the rest are in a tar file that is `scp`'d from a boot server (note that the boot server's ssh public keys are stored in the ramdisk).

The procedure for constructing the ramdisk kernel is similar to the creation of the installation floppy or CDROM that is part of the OpenBSD release.

The configuration tar file is expanded over the skeleton `/etc` directory, thus the machine-specific information overwrites the any common configuration.

The network boot was possible because the machines we use as Ethernet Speakers use the PXE network boot protocol. We found that the network boot procedures for the i386 platform were not sufficiently explained in the OpenBSD documentation, so we produced a netboot "How To" document that is included in our distribution. This document has been submitted to the OpenBSD project for possible inclusion in the project's documentation.

3 Implementation

The virtual audio driver currently runs under OpenBSD 3.4, but we believe that since the audio subsystems between the various BSD-derived Operating Systems are quite similar, porting the VAD to NetBSD and perhaps FreeBSD will not be too difficult.

In this section we discuss some of the technical issues we had to address in developing the VAD and the Audio Steam Rebroadcaster application.

3.1 Rate Limiter, or why does a 5 minute song take 5 minutes?

In a conventional system involving actual audio hardware, the producer-consumer relationship established between the driver and the hardware itself is inherently rate limited. If a five second audio clip is sent to the sound device than it will take five seconds for the actual hardware to play that sound clip. The audio driver cannot send audio data down any faster than this. If an application tries to write to the audio device at rate faster than the hardware can play, it will eventually fill up the ring buffer and the call will begin to block until space is freed.

However, with the VAD there is no underlying hardware to impose a data rate limit. Depending upon the users needs this can be either useful or troublesome. For the purposes of the Ethernet Speaker this creates a serious problem. Without any rate limiting the rebroadcaster will send data that it receives from the VAD as fast as it is written. Assuming that the rebroadcaster is receiving an audio stream from a an MP3 player, then the only speed constraint would be the speed of the I/O and the processor. The producer will essentially send the entire file at wire speed causing the buffers on the Ethernet Speakers to fill up, and the extra data will be discarded, resulting in noticeable audio quality loss. In the above example of the MP3 player you will only hear the first few seconds of the song.

The solution is to instruct the rebroadcaster to sleep for the exact duration of time that it would take to actually play the data, we will effectively limit the rate enough to ensure that it cannot be sent faster than it can be played. The actual duration of this sleep is calculated using the various encoding parameters such as the sample rate and precision.

While we could have integrated this rate limiting into the driver itself we decided against that and developed it separately into the Audio Rebroadcast application. We did not want to limit the functionality of the VAD by slowing it down unnecessarily. Other uses for the VAD might require different needs; we did not wish to make these decisions for the user.

3.2 Synchronization

With the original goal being the placement of any number of Ethernet Speakers on a LAN, issues of synchronizing the playback of a particular audio stream arise. As an Ethernet Speaker accepts data from a stream it needs to buffer the data in order to handle the occasional network hiccup or extraneous packet. It is this buffering that causes problems in synchronizing the playback of the same stream on two different Ethernet Speakers. In earlier versions of the system this problem was most severe when ESs were started at different times in the middle of the stream.

The solution that we implemented is fairly straightforward. Inside, each periodic stream control packet we place a timestamp that serves as a wall clock for the ESs. In addition to this "producer time," we send a timestamp within each audio data packet that instructs the ES when it should play the data. The wall clock and the audio data packet's timestamp are relative to each other, thus allowing the ES to know whether it is playing the stream too quickly or slowly. With this information we can adjust accordingly by either sleeping until it is time to play or throwing away data up until the current wall time. It is important to note however that it is necessary to provide an epsilon value that provides the ES with some leeway. If this is not done than data will be unnecessarily thrown out and skipping in playback will be noticeable.

The aim of this implementation was to ensure that synchronization issues with the various Ethernet Speakers would not be audible. This looser synchronization allowed us to make certain assumptions. Firstly, we completely ignore any latency in the transmission of our packets, essentially assuming that transmission delays over a LAN are uniform (*i.e.,* that everybody receives a multicast packet at the same time). For example, consider a scenario in which one ES receives the wall clock synchronization slightly after another. Both ESs will believe that they have the correct time and will thus play their data, resulting in one being behind the other. Also, there is the issue of slight phase differences that could develop when two ESs have different hardware configurations (*e.g.,* processor speed, audio hardware implementation, and so on). Such changes affect the time consumed by the audio decompression, but more crucially the time delay between writing a block to the audio device and the corresponding sound coming out from the speaker. One could imagine that in combination these two delays may result in serious synchronization problems. However, our initial testing indicates that any phase difference attributed to network delay or otherwise is inaudible.

3.3 VAD implementation

The preliminary design of the VAD device involved a single driver that would attach to the hardware independent audio driver (`dev/audio.c`). The VAD would then be able to intercept and process any audio data written to the audio device file. With full access to this raw audio data, the driver would then send it directly out onto the LAN from within the kernel. The benefits of such a setup were simplicity and efficiency. If all the functionality of the VAD is resident in the kernel then the resulting system is significantly more compact and straightforward. The overall system performance is also slightly better because the kernel-based design avoids the overhead of multiple context switches and the resulting copying of data between kernel and user spaces.

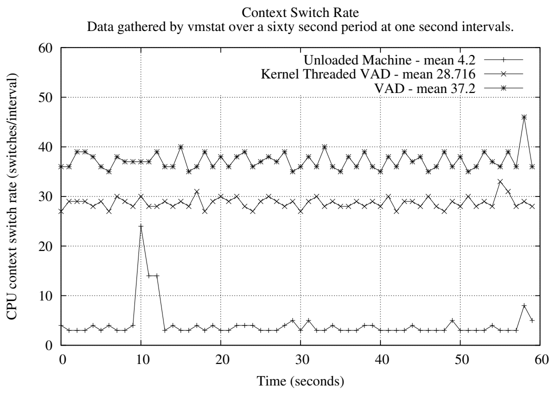


Figure 5: Comparison of context switch rate between a streaming application contained with the VAD driver inside the kernel and a user-level application.

However, in the course of implementation we encountered problems with this design. Since we are developing in kernel space the code must remain relatively simple. If we decided to add complexity to the code in the form of off the shelf compression or security, we would run into problems. Consequently, it became apparent to us as we began developing the VAD that it might be a good idea to separate the streaming functionality from the actual driver itself to make it more modular. Essentially the VAD is supposed to provide a way for us to access the uncompressed audio data written to it so that we can be unconcerned with the prior format it was in. It is our goal to stream this audio data out over a LAN to multiple clients. However, if we separate the streaming functionality from the driver than we allow it to be used for any purpose. With a virtual audio device configured in a system, any application can now have access to uncompressed audio, irrespective of the original format of the audio. By putting only the data access functionality into the driver we have made an addition to the kernel that gives any user space application access to the audio stream. In this way, applications may be developed to process the audio stream (*e.g.,* time-shifting Internet radio transmissions).

Despite our original concerns, relocating the streaming component in user space, does not introduce significant overheads. In Figure 5 we see that introducing a user-level application for data streaming is more expensive, but not significantly so. Moreover, this overhead is swamped by the cost of compression (Figure 5 shows uncompressed streams).

More importantly, there was an interesting aspect of the OpenBSD audio driver architecture that made implementing this design difficult as well. When data is written to the device it enters the hardware independent audio layer and is stored in a ring buffer. For the first block of data ready to be played, the independent audio driver invokes the hardware specific driver attached to it. Then, in the typical example, it is the job of the hardware specific driver to initiate a direct producer-consumer relationship between the hardware and the independent audio driver. Usually this is done by triggering a DMA transfer of the data to the actual hardware. The hardware specific driver passes an interrupt function, provided by the independent audio driver, to the DMA routines that is called every time a transfer is completed, usually by the audio hardware interrupt service routine. This interrupt function notifies the hardware independent driver that a block of data has been transferred and can be discarded. With this relationship intact, the hardware specific driver is essentially out of the picture, cutting out the middleman so to speak. Therefore it is only invoked once, when the first block of data is ready to be played.

The problem arises when there is no actual hardware to create such a relationship with. Our VAD driver takes the place of the hardware specific driver however there is no actual sound hardware. So when the first block of data is written the independent audio driver assumes that the VAD driver will set up the DMA and pass along the interrupt function. In our case, the independent audio driver assumes the VAD is just the middleman, expecting it's interrupt routine to be called by the hardware, it never invokes the VAD entry points again. A strong example of the fact that the OpenBSD audio architecture was not meant to support pseudo devices. Our solutions to this problem were inelegant and involved either modifying the independent audio driver or creating a separate kernel thread to periodically call the interrupt routine.

3.4 Platform

The machines we used for the Ethernet speaker reflect our belief that a mass produced low-cost Ethernet Speaker platform will be relatively resource poor. We have, therefore, used Neoware EON 4000 machines that have a National Semiconductor Geode processor running at 233MHz and 64Mb RAM, non-volatile memory (Flash) and built-in audio and Ethernet interfaces (Figure 6).

While weak compared to the capabilities of current workstations, the hardware is perfectly adequate for the intended application. Moreover, the use of such low-cost hardware will allow the Ethernet Speaker to have a cost

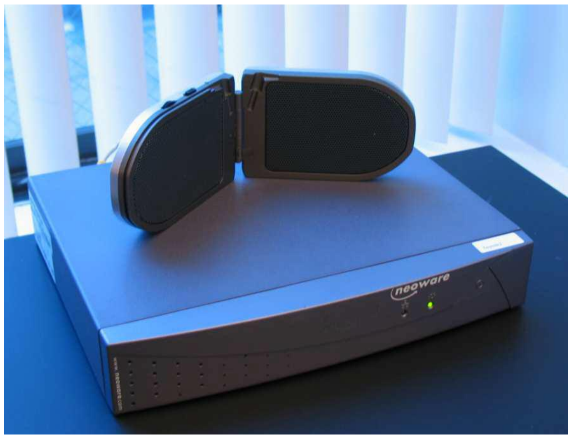


Figure 6: The Ethernet Speaker is based on the Neoware EON 4000.

of less the $50.

Our test environment also included a SUN Ultra 10 workstation in order to make sure that our programs and communication protocol worked across platforms.

The slow speed of the processor on the EON 4000 computer, revealed a problem that was not observed during our testing on faster machines; namely the need to keep the pipeline full. If we use very large buffers, the decompression on the ES has to wait for the entire buffer to be delivered, then the decompression takes place and finally the data are fed to the audio device at a rate dictated by the audio sampling rate. If the buffers are large, then time delays add up, resulting in skipped audio. By reducing the buffer size, each of the stages on the ES finishes faster and the audio stream is processed without problems.

4 Related Work

4.1 Audio Streaming Servers

SHOUTcast [1] is a MPEG Layer 3 based streaming server technology. It permits anyone to broadcast audio content from their PC to listeners across the Internet or any other IP based network. It is capable of streaming live audio as well as on-demand archived broadcasts.

Listeners tune in to SHOUTcast broadcasts by using

[1] http://www.shoutcast.com/support/docs/

a player capable of streaming MP3 audio e.g.Winamp for Windows, XMMS for Linux etc. Broadcasters use Winamp along with a special plug-in called SHOUTcast source to redirect Winamp's output to the SHOUTcast server. The streaming is done by the SHOUTcast Distributed Network Audio Server (DNAS). All MP3 files inside the content folder are streamable. The server can maintain a web interface for users to selectively play its streamable content.

The Helix Universal Server from RealNetworks [2] is a universal platform server with support for live and on-demand delivery of all major file formats including Real Media, Windows Media, QuickTime, MPEG4, MP3 and more. It is both scalable and bandwidth conserving as it comes integrated with a content networking system, specifically designed to provision live and on-demand content. It also includes server fail-over capabilities which route client requests to backup servers in the event of failure or unexpected outages.

4.2 Audio Systems

There are also a number of products that may be described as "internet radios". The most well known is Apple's AirTunes for its Airport Express base station. Such devices accept audio streams generated by servers either in the LAN or in the Internet and play it back. Their feature sets and capabilities are very similar to those of the ES, with the exception that (at least the ones we have tested) they do not provide synchronization between nearby stations. As such they may not be able to be used in an ES context where we have multiple synchronized audio sources within a given room.

4.3 Network Audio Redirectors

The Network Audio System [8] (NAS) developed by NCD uses the client/server model for transferring audio data between applications and desktop X terminals. It aims to separate applications from specific drivers that control audio input and output devices. NAS supports a variety of audio file and data formats and allows mixing and manipulating of audio data.

Similar to NAS are audio servers such as Gstreamer [14] and aRts [5] that allow multiple audio sources to use the workstation's audio hardware.

[2]https://www.helixcommunity.org/2002/intro/client

The most sophisticated of the lot is JACK [3], which is a low latency audio server.

All the above are constrained by the fact that they need to either have new applications created for them, or at least to recompile existing applications with their own libraries. Obviously this works only with source distributions.

EsoundD [2] utilizes the fact that most applications use dynamically-linked libraries and uses the LD_LIBRARY_PATH variable to insert its own libraries before the system ones. The applications (apparently the Real Audio Player is included) use the EsoundD libraries and thus benefit from the audio redirection and multiplex features without the need for any recompilation. Of course, if the application is statically linked (not very popular lately), this approach fails.

The closest system to the VAD is the maudio [1] virtual audio device for Linux. The maudio provides functionality that is very close to that of the VAD component of our Ethernet Speaker System. Maudio also provides a virtual driver for the audio device for the Linux kernel and we have used it to port our audio rebroadcaster service to Linux.

A similar application to the VAD is the Multicast File Transfer Protocol [6] (MFTP) from StarBurst Communications. MFTP is designed to provide efficient and reliable file delivery from a single sender to multiple receivers.

MFTP uses a separate multicast group to announce the availability of data sets on other multicast groups. This gives the clients a chance to chose whether to participate in an MFTP transfer. This is a very interesting idea in that the client does not need to listen-in on channels that are of no interest to it. We plan to adopt this approach in the next release of our streaming audio server, for the announcement of information about the audio streams that are being transmitted via the network. In this way the user can see which programs are being multicast, rather than having to switch channels to monitor the audio transmissions.

Another benefit from the use of this out-of-band catalog, is that it enables the server to suspend transmission of a particular channel, if it notices that there are no listeners. This notification may be handled through the use of the proposed MSNIP standard [7]. MSNIP allows the audio server to contact the first hop routers asking whether there are listeners on the other side. This allows the server to receive overall status information without running the risk of suffering the "NAK implosion" problem. Unfortunately, we have to wait until the MSNIP appears in the software distributions running on our campus routers.

5 Future Plans

5.1 Security

For the Audio Steam Rebroadcaster, security is important. The ESs want to know that the audio streams they see advertised on the LAN are the real ones, and not fake advertisements from impostors.

Moreover, we want to prevent malicious hosts from injecting packets into an audio stream. We do this by allowing the ES to perform integrity checks on the incoming packets.

In the current version some security can be maintained by operating the Ethernet Speakers in their own VLAN. Note, however, that there exist ways for injecting packets into VLANs, so this must be considered as an interim measure at best.

Our basic security requirements are that (a) the ES should not play audio from an unauthorized source, and (b) the machine should be resistant to denial of service attacks.

Having a machine that receives all its boot state from the network creates an inherently unsafe platform. Any kind of authentication that is sent over the network may be modified by a malicious entity, thus creating the environment for the installation of backdoors that may activated at any moment.

We are considering taking advantage of the non-volatile RAM on each machine to store a Certification Authority key that may be used for the verification of the audio stream.

For the audio authentication digitally signing every audio packet [15] is not feasible as it allows an attacker to overwhelm an ES by simply feeding it garbage. We are, therefore, examining techniques for fast signing and verification such as those proposed by Reyzin et al [13], or Karlof et al [9]. We are also looking into whether we can take advantage of the services offered by the IEEE 802.1AE MAC-layer security standard.

5.2 Automation

Another area where we are working on is the ability for the device to perform actions automatically. One example will be to set the volume level automatically depending on the ambient noise level and the type of audio stream. So for background music the ES would lower the volume if the area is quiet while ensuring that audio segments recorded at different volume levels produce the same sound levels.

Alternatively, if an announcement is being made, then the volume should be increased if there is a lot of background noise so that announcements are likely to be heard.

We plan to implement these features by taking advantage of the microphone input available on our machines. This input allows the ES to compare its own output against the ambient levels.

5.3 Management

Our intention is to have multiple streams active at the same time and ESs being able to switch from one channel to another. This implies the ability to receive input from the user (*e.g.,* some remote control device). Alternatively all ESs within an administrative domain may need to be controlled centrally (*e.g.,* movies shown on TV sets on airplane seats can be overridden by crew announcements). We want to investigate the entire range of management actions that may be carried out on ESs and create an SNMP MIB to allow any NMS console to manage ESs.

6 Conclusions

Existing audio players adopt complex protocols and elaborate mechanisms in order to deal with network problems associated with transmission over WAN links. These players are also largely incompatible with each other while the use of unicast connections to remote servers precludes the synchronization of multiple players within the same locality. Moreover, these multiple connections increase the load both on the remote server and on the external connection points of the network and the work that has to be performed by firewalls, routers etc. Finally, the large number of special purpose audio players (each compatible with a different subset of available formats), alienates users and creates support and administrative headaches.

By implementing the audio streaming server as a virtual device on the system running the decoding applications, we have bypassed the compatibility issues that haunt any general-purpose audio player. Our system confines the special-purpose audio players to a few servers that multicast the audio data always using the same common format.

The existence of a single internal protocol without special cases or the need for additional development to support new formats, allowed the creation of the Audio Rebroadcast Application that allows clients to play audio streams received from the network. The communications protocol also allows any client to "tune" -in or -out of a transmission, without requiring the knowledge or cooperation of the server.

The software for the VAD and Audio Rebroadcasting application is available as Open Source Software and can be downloaded at `http://www.cs.drexel.edu/~vp/EthernetSpeaker`.

References

[1] A simple audio mirroring device . `http://freshmeat.net/projects/maudio`.

[2] EsounD The Enlightened Sound Daemon . `http://www.tux.org/~ricdude/dbdocs/book1.html`.

[3] JACK: A Low Latency Audio Server. `http://jackit.sourceforge.net`.

[4] QSControl CM16a Amplifier Network Monitor Product Information Sheet. `http://www.qscaudio.com/pdfs/qsconspc.pdf`.

[5] The Analog Real-Time Synthesizer. `http://www.arts-project.org/doc/manual`.

[6] K. Miller et all. StarBurst multicast file transfer protocol (MFTP) specification. Internet Draft, Internet Engineering Task Force, April 1998.

[7] Bill Fenner, Brian Haberman, Hugh Holbrook, and Isidor Kouvelas. Multicast Source Notification of Interest Protocol (MSNIP). draft-ietf-magma-msnip-01.txt, November 2002.

[8] Jim Fulton and Greg Renda. The network audio system: Make your applications sing (as well as dance)! *The X Resource*, 9(1):181–194, 1994.

[9] C. Karlof, N. Sastry, Y. Li, A. Perrig, and J. Tygar. Distillation codes and applications to dos resistant multicast authentication, 2004.

[10] Ishan Mandrekar, Vassilis Prevelakis, and David Michael Turner. An Audio Stream Redirector for the Ethernet Speaker. In *International Network Conference*, Plymouth, UK, July 2004.

[11] V. Prevelakis and A. D. Keromytis. Drop-in Security for Distributed and Portable Computing Elements. *Internet Research: Electronic Networking, Applications and Policy*, 13(2), 2003.

[12] Vassilis Prevelakis. A Secure Station for Network Monitoring and Control. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[13] Leonid Reyzin and Natan Reyzin. Better than BiBa: Short one-time signatures with fast signing and verifying. In *Seventh Australasian Conference on Information Security and Privacy (ACIP 2002)*, July 2002.

[14] Christian F.K. Schaller and Scott Wheeler. Introduction to GStreamer & KDE. `http://conference2004.kde.org/slides/schaller.wheeler-introduction_to_g%streamer_and_kde.pdf`.

[15] Wong and Lam. Digital signatures for flows and multicasts. *IEEETNWKG: IEEE/ACM Transactions on NetworkingIEEE Communications Society, IEEE Computer Society and the ACM with its Special Interest Group on Data Communication (SIGCOMM), ACM Press*, 7, 1999.

A PC-Based Open-Source Voting Machine with an Accessible Voter-Verifiable Paper Ballot

Arthur M. Keller, UC Santa Cruz and Open Voting Consortium
ark@soe.ucsc.edu
Alan Dechert, Open Voting Consortium
alan@openvotingconsortium.org
Karl Auerbach, InterWorking Labs
karl@iwl.com
David Mertz, Gnosis Software, Inc.
mertz@gnosis.cx
Amy Pearl, Software Innovations
amy@swinvent.com
Joseph Lorenzo Hall, UC Berkeley SIMS
jhall@sims.berkeley.edu

1. Introduction

Voting is the foundation of a democratic system of government, whether the system uses direct or representative governance. The heart of voting is trust that each vote is recorded and tallied with accuracy and impartiality. There is no shortage of historical examples of attempts to undermine the integrity of electoral systems. The paper and mechanical systems we use today, although far from perfect, are built upon literally hundreds of years of actual experience.

There is immense pressure to replace our "dated" paper and mechanical systems with computerized systems. There are many reasons why such systems are attractive. These reasons include, cost, speed of voting and tabulation, elimination of ambiguity from things like "hanging chads," and a belated recognition that many of our traditional systems are not well suited for use by citizens with physical impairments.

However, electronic voting brings a new set of risks and drawbacks as well as advantages. In response to the problems and opportunities of electronic voting, the Open Voting Consortium was established.

The Open Voting Consortium (OVC) is creating an open source, trustworthy, cost effective, voter verifiable voting system using open source software components on industry standard computers. A primary element of this Open Voting system is the use of software through which the voter creates a *printed paper ballot* containing his or her choices. Before casting his or her ballot the voter may use other, independently programmed, computers to verify that the ballot properly reflects the voter's choices. The voter may also visually inspect the text printed on the paper ballot. The paper ballot is cast by placing it into a ballot box. Once cast, that paper ballot is the authoritative record of the voter's choices for the election and for any recount of that election. Open Voting ballots are machine-readable and may be tabulated (and re-tabulated in the case of a recount) either by computer or by hand.

Open Voting systems can be engineered to accommodate the special needs of those who have physical impairments, or limited reading ability.

Many of us today have come to trust many of our financial transactions to ATMs (automatic teller machines). The push for electronic voting machines has benefited from that faith in ATMs. However, we are starting to learn that that faith is unwarranted.

First, ATM machines do fail and are often attacked. Those who operate ATMs usually consider the loss rate to be a proprietary secret. Banks are well versed in the actuarial arts and they build into their financial plans various means to cover the losses that do occur. In more crude terms, it's only money.

Second, voting machines carry a more precious burden — there is no way to buy insurance or to set aside a contingency fund to replace a broken or tampered election.

There are several areas of concern regarding the new generation of computerized voting machines:

- No means for the voter to verify that his or her votes have been tallied properly.
- No means outside of the memories of the voting machines themselves to audit or recount the votes.
- Lack of ability to audit the quality of the software. Fortunately the widespread belief that "computers are always right" is fading. Our individual experiences with error-ridden software on personal computers and consumer products (e.g., the BMW 745i[1]), software errors

by even the best-of-the-best (e.g., NASA and the loss of the Mars Climate Orbiter[2]), and the possibility that intentional software bugs can be hidden so deeply as to be virtually invisible (Ken Thompson's famous 1984 paper — Reflections on Trusting Trust[3]) have all combined to teach us that we should not trust software until that trust has been well earned. And even then, we ought not to be surprised if unsuspected flaws arise.

- Vulnerability of the machines or of their supporting infrastructures to intentional attack or inadvertent errors.

The companies that produce voting machines have poured gasoline onto the smoldering embers of concern. Some of these products are built on Microsoft operating systems — operating systems that have a well-earned reputation for being penetrable and unsecure.[4] And most of these companies claim that their systems are full of trade secrets and proprietary information and that, as a consequence, their internal workings may not be inspected by the public. In addition, these companies have frequently displayed a degree of hostility (in some cases, the hostility took the form of lawsuits[5]) against those who are concerned about the integrity of these products. And finally, their systems have "vulnerabilities [that] do show incompetence" and were built by "programmers [who] simply don't know how to design a secure system."[6] There is a widespread perception that these companies are more concerned about profits than about fair and trustworthy elections.

The Help America Vote Act of 2002[7] was passed into law to modernize voting equipment as a result of the 2000 US Presidential election and the problems observed in Florida.[8] The Federal Election Commission (FEC) has issued a set of Voting System Standards (VSS)[9] that serve as a model of functional requirements that elections systems must meet before they can be certified for use in an election. The next section discusses the existing voting machines that meet those standards. Section 3 considers the rationale for an accessible voter-verifiable paper ballot. Section 4 is a description of the Open Voting Consortium architecture for the polling place. Section 5 mentions the current state of the system and next steps for its development. Conclusion, acknowledgements, and references follow.

2. OVC System Description

The OVC system will be very much like a traditional system in which the voter enters the polling place, marks his or her choices onto a paper ballot, and inserts the ballot into a ballot box. Our design applies computer technology to that traditional system. However, unlike some of the other computerized voting systems that change the basic nature of the traditional system, our design applies computer technology only in a limited and conservative way.

The OVC design preserves the paper ballot. However, under the OVC design the voter marks the ballot using a computerized voting station rather than a pencil or colored marker. The ballot is printed in plain text that the voter can read. Voters have the opportunity to inspect the ballot to ensure that it properly reflects their choices.

The OVC design will preserve the ballot box. Voters must insert their paper ballots into the ballot box. The OVC ballots will contain a barcode in addition to the plain text. This barcode makes it easy for the poll workers to count the ballots[10] when the ballot box is opened.

The OVC design will be a voter-verified voting system. The core difference between this and other systems, such as DRE equipped with printers, is that in the OVC design the paper ballot is the actual ballot; information that might be recorded in computer memories or on computer media is used only for security, error-detection, fraud detection/prevention, and auditing.

3. Existing Electronic Voting Machines

Existing DRE (Direct Recording Electronic) voting machines have come under increasing scrutiny with widespread reports of malfunctions, omissions and user interface problems during elections.

3.1 Diebold AccuVote TS and TS-X

A group led by Avi Rubin and Dan Wallach analyzed the Diebold AccuVote TS DRE voting machine and found numerous flaws.[11] SAIC was commissioned by the state of Maryland to analyze the Diebold voting system and found "[t]he system, as implemented in policy, procedure, and technology, is at high risk of compromise."[12] Based on these reports, the California Secretary of State's office established security procedures for DRE voting machines.[13] Diebold was then found to have used uncertified software in 17 counties in California.[14] The California Secretary of State then decertified the Diebold and all other DREs on April 30, 2004.[15]

3.2 Electronic Systems and Software iVotronic

ES&S iVotronic is a poll-worker-activated, multilingual touch screen system that records votes on internal flash memory. A poll worker uses a cartridgelike device called a Personal Electronic Ballot (PEB) to turn the machine on and enable voting. Voters first choose their ballot language and then make their selections via a touch screen. When the polls close, poll workers read summary data from each machine onto the PEB via infrared. The PEBs are then transported to election headquarters or their contents transmitted via a computer network.

In September 2002 in Florida, a spot check of iVotronic machines revealed several precincts where hundreds of voters had only one or even no selections chosen the their ballots cast on Election Day and vote totals produced by the main and backup system did not agree.[16] In October 2002 in Texas, several people reported that their votes registered for a different candidate on screen and, in fact, some votes cast for Republicans were counted for Democrats.[17] In November 2002, two early-voting locations in Wake County, North Carolina (Raleigh) failed to record 436 ballots due to a problem in the iVotronics' firmware.[18]

3.3 Hart InterCivic eSlate

Hart's eSlate is a voter-activated multilingual voting system where the voter turns a selector wheel and set of buttons to indicate their votes. The eSlate terminals are connected via daisy-chained serial cable to a central controller, the Judges' Booth Controller (JBC), which provides power, vote activation, and vote storage for up to twelve eSlate terminals. A poll worker issues a 4-digit PIN to the voter using the JBC. The voter enters this PIN on an eSlate and votes using its selector wheel and buttons. Once the vote is cast, the vote is transmitted via a cable to the JBC and stored in flash memory on the JBC's Mobile Ballot Box (MBB). The MBB is then either physically transported to election headquarters or its contents transmitted via computer network.

In November 2003, poll workers in Harris County, Texas, confused by the system's complexity, could not get the machines to work properly and had been assigning the wrong ballots to voters using the JBC.[19] In February 2004 in Virginia, voters had to cast paper ballots when the JBC used at one precinct "fried," rendering all the eSlate machines unusable.[20] In March 2004 in Orange County, California, hundreds of voters were turned away when one eSlate machine broke down.[21] At the same time in California, poll workers incorrectly assigned ballots from different precincts to their voters and approximately 7000 voters cast ballots for the incorrect precinct.[22]

3.4 Sequoia Voting Systems AVC Edge

The Sequoia AVC Edge is a voter-activated multilingual touch screen system that records votes on flash memory. Its operation is very similar to the Diebold AccuVote-TS described above.

In March 2002 in Palm Beach County, Florida, the Edge machines froze up when voters selected their ballot language and other reports indicate votes counted for the wrong candidate.[23] As well, 15 PCMCIA cards were temporarily lost and the central system would not report the results and in a very close race many ballots were blank.[24] In April 2002, in Hillsborough County, Florida, one precinct could not transfer data on 24 out of 26 PCMCIA cards; results were faxed and entered in by hand.[25] In March 2003, a similar problem plagued PCMCIA cards.[26] In November 2002, in Bernalillo County, New Mexico, 48,000 people voted early, but no race showed more than 36,000 votes due to a software bug.[27] In June 2004, in Morris County New Jersey, the central tabulation system read only zeros from the PCMCIA cards.[28]

3.5 Other DRE Voting Machines

Other DRE vendors are proposing to add printers to their DREs.[29] AccuPoll has an Electronic Voting System with a voter-verified paper audit trail[30] and Sequoia Voting Systems is marketing optional voter-verified paper record printers for their DREs.[31] The state of Nevada used these VeriVote printers in the 2004 primary and presidential elections.[32] Although some predicted that the printers would fail or cause long lines, "[t]he primary election was free of serious problems that have embarrassed registrars in Florida, California, Maryland and other states with touchscreen machines."[33] The Avante Vote-Trakker is a DRE with a voter-verified paper audit trail.[34] However, none of these systems are in wide use, and some have not even been certified. None of them meet the current California AVVPAT standards.

3.6 Paper Ballots with Optical Scan Machines

There are several problems with the use of paper ballots that are optically scanned with mark/sense-type tabulation systems. The paper ballot is not accessible to the visually impaired or reading impaired. And the paper ballot must be available in multiple languages as required by the jurisdiction. The use of paper does enable recounts, but potentially suffers from the problems of overvotes, undervotes, and improper changes to ballots (including extraneous marks, which would void the ballot).

3.6.1 AutoMark

The AutoMark[35] system is an Electronic Ballot Marker (EBM) that addresses accessibility problems by using an interface comparable to a Direct Recording Electronic voting machine. Similarly, it can provide support for multiple languages and limit overvotes and undervotes through its user interface. AutoMark uses ballots identical to those also used for manually-marked optical-scan systems. AutoMark effectively replaces the pen in such systems with a marking device that supports multiple languages and detects overvotes and undervotes. So there is the question of whether the printed ballots are in each required language, so that a non-English-speaking voter can still verify his or her ballot. It is possible to have a device with accessible output modes that reads the voter's marked choices (like that of OVC system described in Section 5.1.7), so that the voter may verify that the ballot is marked correctly, but that is not currently part of the AutoMark system. A key benefit of the AutoMark system is that the same optical scan tabulation system can be used for ballots cast in polling places, absentee ballots, and provisional ballots. But they neither maintain an electronic audit trail nor use digital signatures to detect ballot stuffing.

However, the optical scan tabulating systems are also potential sources of error. They must be calibrated to properly distinguish the ballot markings. For example, during the March 2004 primary election in Napa County, California, "optical scan machines from Sequoia Voting Systems failed to record voters' ballot marks on the paper copy.... Apparently, the optical scan machines can read carbon-based ink but not gel pens."[36] There is also the potential that the vote tabulation software could have errors or have been fraudulently modified.

3.6.2 Populex

The Populex[37] system is an Electronic Ballot Printer (EBP) that also addresses accessibility by using an interface comparable to a Direct Recording Electronic Voting Machine. The output is a small printed ballot with a barcode that contains the voter's ballot selections. Numbers are printed on the bottom of the ballot so that the voter can read the choices, but these numbers must be matched to a chart (e.g., on the wall) in the precinct. Because the ballots are paper, a provisional ballot could be placed in an envelope for subsequently determining whether the ballot should be counted. However, absentee ballots must be counted using a different technology, unless the absentee voter uses a Populex system.

3.7 DREs used internationally

Various countries around the world have chosen to use DRE systems. The Dutch-based NEDAP[38] system is used in the Netherlands, France, Germany, the United Kingdom and Ireland but has been criticized for its flaws.[39] The voting machine developed and used recently in India[40] is very simple and lacks features like disabled and multilingual access. Venezuela chose VVPAT-enabled Smartmatic[41] DREs on which to conduct its recent referendum on the President; there were allegations of fraud, but the "audit concluded the voting machines did accurately reflect the intent of the voters, as evidenced by a recount of the paper ballots in a sample of the machines."[42]

4. Why an Accessible Voter-Verifiable Paper Ballot

Many computer and other experts have joined VerifiedVoting.org's call for "the use of voter-verified paper ballots (VVPBs) for all elections in the United States, so voters can inspect individual permanent records of their ballots before they are cast and so meaningful recounts may be conducted. We also insist that electronic voting equipment and software be open to public scrutiny and that random, surprise recounts be conducted on a regular basis to audit election equipment."[43]

4.1 Paper Receipts vs. Paper Ballots

We speak of OVC creating a paper *ballot*, not a receipt, nor simply a "paper trail." That is, for OVC machines, the printout from a voting station is the primary and official record of votes cast by a voter. Electronic records may be used for generating preliminary results more rapidly or for auditing purposes, but the paper ballot is the actual official vote document counted.[44]

Some writers discuss producing a paper receipt, which a voter might carry home with them, as they do an ATM receipt. There are two significant problems with this approach. In the first place, if we suppose that a voting station might have been tampered with and/or simply contain a programming error, it is not a great jump to imagine that it may print out a record that differs from what it records electronically. A receipt is a "feel good" approach that fails to correct the underlying flaws of DREs.

But the second problem with receipts is even more fundamental. A voting receipt that can be carried away by a voter enables vote buying and vote coercion. An interested third party—even someone as seemingly innocuous as an overbearing family member—could demand to see a receipt for voting in a manner desired. With OVC systems, ballots must be placed into a sealed ballot box to count as votes. If a voter leaves with an uncast ballot, even if she went through the motions of printing it at a vote station, that simply does not represent a vote that may be "proven" to a third party.

What some vendors refer to as a paper trail suffers from a weakness similar to the first problem paper receipts suffer. Under some such models, a DRE voting station might print out a summary of votes cast at the end of the day (or at some other interval). But such a printout is also just a "feel good" measure. If a machine software or hardware can be flawed out of malice or error, it can very well print a tally that fails to accurately reflect the votes cast on it. It is not *paper* that is crucial, but *voter-verifiability*.

4.2 Paper Audit Trail Under Glass vs. Paper Ballot

While "paper audit trail under glass" does indeed do a pretty good job of preventing ballot box stuffing with forged physical ballots, this approach is not the only—nor even the best—technique to accomplish this goal. We plan for OVC systems to incorporate cryptographic signatures and precinct-level customization of ballots that can convincingly prove a ballot is produced on authorized machines, at the polling place, rather than forged elsewhere. For example, a simple customization of ballots is a variation of the page position of our ballot watermarks in a manner that a tamperer cannot produce in advance. Surprisingly much information can be subtly coded by moving two background images a few millimeters in various directions. Another option is to encode a cryptographic signature within the barcode on a ballot—in a manner that can be mathematically proven not to disclose anything about the individual voter who cast that vote, but simultaneously that cannot be forged without knowledge of a secret key, which is known only to that electronic voting machine.

There are several narrowly technical problems with “paper audit trail under glass” systems. A “paper audit trail under glass” system has some extra mechanical problems with allowing rejection of incorrect paper record; some sort of mechanism for identifying the paper record as spoiled, perhaps through an ink mark. This approach increases the potential of physical failure, such as paper jams.

A more significant issue for "paper audit trail under glass" systems is their failure to provide the quality of accessibility to vision- or reading-impaired voters that OVC's design does. Sighted voters who happen to need reading glasses, or who can read only large print or closely held print, are likely to find “paper audit trail under glass” systems more difficult to check than printed ballots they can physically hold. Even if these machines add provisions for audio feedback on final ballots, users are dependent on the very same machine to provide such audio feedback. Potentially, a tampered-with machine could bias votes, but only for blind voters (still perhaps enough to change close elections). In contrast, OVC positively encourages third parties to develop software to assure the barcode encoding of votes matches the visibly printed votes—every voter is treated equally, and all can verify ballots.

From a more sophisticated cryptology perspective, “paper audit trail under glass” systems are likely to compromise voter anonymity in subtle ways. One of the issues the world-class security researchers associated with our design have considered is the possibility that sequential or time-stamp information on ballots could be correlated with the activity of individual voters. Even covert videotaping of the order in which voters enter a polling place might be used for such a compromise. This problem is more serious in those systems in which the voter-verified paper audit trail is maintained on a continuous paper tape fed onto a take-up spool. However, on systems that cut the audit trail into pieces, one for each voter, that ballots that fell to the bottom of the glass bin may be visible to subsequent voters. Such potential visibility also compromises anonymity. This analysis is just part of the threat analysis study that we will perform in order to create a reliable, secure, and trustworthy election system.

A far more serious problem with a voter-verified paper audit trail is the difficulty of automated tabulation of the audit trail. This problem is especially acute when the voter-verified paper audit trail is cut into pieces, one for each voter. The glass bin is likely to be a mass of coiled paper strips. While the continuous spool approach to the paper audit trail is neater, it suffers from an anonymity problem as identified above. When the paper audit trail can only be used in a manual tabulation process, there will be enormous pressure to minimize its use, thereby reducing its effectiveness. In contrast, the OVC design facilitates automated tabulation of the paper ballots while enabling manual counting and voter-verification also. However, even when there is an automated tabulation process of the paper ballots, it is critical for a random sample of precincts to be manually tabulated, as a further check on the system.

4.3 Accessible Voting

One of the key benefits of electronic voting machines is to allow disabled voters to vote unassisted.[45] However, as the movement for a voter-verifiable paper audit trail grows,[46] there is a need for the paper audit trail to be accessible as well.[47] The Open Voting Consortium's voting system is designed to be accessible for both entering the votes and verifying the paper ballot produced.

5. OVC System Overview

The Open Voting Consortium (OVC) is developing a PC-based open source voting machine with an accessible voter-verified paper ballot. We intend to use an open source operating system for the PC, such as Knoppix, a variant of Linux that boots off of a CD. The polling place system consists of a Voter Sign-in Station, an Electronic Voting Station, an Electronic Voting Station with a Reading-Impaired Interface, a Ballot Verification Station, and a Ballot Reconciliation Station. See Figure 1 (at end of paper). In addition, there are components at the county canvassing site that are discussed only briefly in this paper.

5.1 Precinct/Polling Place Element

The OVC Precinct/Polling Place Element is intended to provide all of the systems and procedures required for a polling place except for voter rolls, sign-in books and the like. The OVC system will be flexible so that it will be adaptable to applicable laws as well as local preferences.

The OVC design will accommodate polling places in which different classes of voters, for example voters of different parties, may be accommodated with ballots appropriate for that particular voter.

Overall design and operation of the OVC system is simplified because the paper ballot produced by it will be the legal ballot. For example, in the OVC design equipment failures will not be handled at the polling place except to the extent necessary to disconnect the failed unit, seal it, and deploy a backup unit.

The OVC design is very concerned that voters with physical disabilities and limited reading ability are accommodated. It is anticipated that the OVC design will contain variations of the Voting Station and Ballot Verification Station that will be designed with user interfaces tailored to the needs of voters with certain types of physical impairments.

The OVC design will not require substantial changes to the workflow of a typical polling place; voters and poll workers will find that procedures are comparable to those used in existing American polling places.

5.1.1 Voter Sign-in Station

The Voter Sign-In Station is used by the poll worker when the voter signs in and involves giving the voter a "token." It is a requirement that each voter cast only one vote and that the vote cast be of the right precinct and party for the voter. The "token" authorizes the voter to cast a ballot using one of these techniques.

- Pre-printed ballot stock
  - Option for scanning ballot type by EVM
- Poll worker activation
- Per-voter PIN (including party/precinct identifier)
- Per-party/precinct token
- Smart cards

The token is then used by the Electronic Voting Station and the Electronic Voting Station with the Reading Impaired Interface to ensure that each voter votes only once and only using the correct ballot type.

If the voter spoils a ballot, the ballot is marked spoiled and kept for reconciliation at the Ballot Reconciliation Station, and the voter is given a new token for voting.

In the case of a per-voter PINs or smart cards, it is imperative that these do not personally identify the voter.

5.1.2 Electronic Voting Station

The Voting Station is the voter's primary point of contact with the OVC system. After the voter signs-in, a poll worker will direct the voter to a Voting Station.

The physical appearance of the voting station will be that of a lightweight booth with privacy curtains or walls. There will be an integrated device—an Electronic Voting Station—containing computer, printer, battery, and flat screen display. The display will allow touch-screen use and will be mounted so that it may be adapted for use by voters who stand and voters who are in wheel chairs.

The Voting Station will be designed so that setup and teardown are easy; it is anticipated that installation will be largely an unfold-and-plug-in operation.

The Voting Station will be tamperproof and be engineered to endure physical abuse during shipping, deployment, and use. The Voting Station will be designed so that it may be sealed against unauthorized access with locks and lead/wire seals.

We are developing procedures that ensure that the correct certified software is operating on the voting machines, from booting from a CD rather than the hard drive, verifying checksums and hardware configurations, as well as other means.

The Electronic Voting Station consists of these components:

- A computer, preferably stock commodity hardware, with these features:
  - A monitor, preferably LCD, possibly 15" or 17" touch-screen measured diagonally.
  - One or more input devices, such as:
    - Touch-screen interface on LCD screen
    - Mouse
    - Keyboard
    - Buttons surrounding the screen, like on an ATM
    - Numeric keypad
    - Symbolic keypad
  - Possibly a smart card reader/writer
- A CD-R drive. The CD-R will contain:
  - The operating system, e.g., a Linux system without unnecessary components
  - The EVM software
  - Ballot Definition files and public keys of various external components
  - Optionally, sound files for the ballot (included for the Electronic Voting Station with the Reading Impaired Interface)
  - Personalization, potentially including public/private key pairs[48] for this voting station
  - Startup record, possibly including generated public key of this voting station
  - Electronic Ballot Images (EBIs), in XML format (and possibly in Postscript format), written at end of day in ascending order by (randomly generated) ballot ID
  - The CD-R is used subsequently by the Ballot Reconciliation System and possibly during county canvassing.
  - A printer with these specifications:
  - Inkjet or laser
  - Preferably output page is obscured from view (either by appearing face down, or by a cover)
  - Feedback to the user (auditory or visual) that the ballot is printing and will come out soon
  - Prints a test document at the start of a voting day that includes records of the public keys for the EVM for this day.
  - Depending on voter "token," EVM takes blank ballot stock given to voter upon sign-in; alternatively, EVM includes storage for blank ballot stock for printing. Blank ballot stock may be specially printed paper, possibly pre-printed on reverse side (with "please turn over" message).
  - Prints ballot in printed ballot format potentially using special printed ballot stock.
  - The ballot can be read by the Ballot Verification Station and includes text in OCR format, plus a barcode for more foolproof reading.
  - A persistent EBI storage device, such as a USB memory dongle (i.e., a USB flash memory device) for persistently storing the EBIs until the end of the day, when the EBIs are transferred onto the CD-R. The USB memory dongle is kept for audit purposes.
  - Device should be large enough not to be easily lost
  - Device should be lockable and tamper proof when locked
  - Potentially, device could lock in the open position onto cabinet and PC and lock in the closed position sealed and ready for removal. Device could be set to be open only once, and on subsequent openings the device would be read only.
  - May also have hardware private key for digitally signing the ballot.
  - Security enclosure that prevents tinkering with the device

5.1.3 Electronic Voting Station with Reading Impaired Interface

The Electronic Voting Station with Reading Impaired Interface is a computer similar to the Electronic Voting Station described above that includes auditory output of the ballot choices and selections made and also includes additional modes of making selections suitable for the blind or reading impaired. Whether these features are integrated to a common voting machine with all functionality, or whether there is a separate configuration for the disabled, is an open question. For example, additional modes of input may be useful for those who can read printed materials, but have physical limitations. The ideal is a universal design that accommodates all voters.

The electronic voting station for the reading impaired produces a printed ballot that can be processed by the Ballot Verification Station.

5.1.4 Paper Ballot

The paper ballot is generated by the Electronic Voting Station or the Electronic Voting Station with Reading Impaired Interface. It is the paper on which the voter's choices are recorded. It must be "cast" in order to be tallied during canvassing, testing, or a manual recount.

The paper ballot can easily be read by the voter so that the voter may verify that his or her choices have been properly marked. It also contains security markings and a barcode. The barcode encodes the user's choices, as expressed in the human readable portion of the ballot. The human readable text should be in an OCR-friendly font so it is computer-readable as well. The voter may use the Ballot Verification Station to verify that the barcode accurately reflects their choices. The Ballot Verification Station not only assists sight-impaired and reading-impaired voters in verifying their ballots, but also to give any voter the assurance that the barcode on the ballot properly mirrors their choices, as represented in the human-readable text on the ballot. Ideally, the Ballot Reconciliation System reads the OCR text of the ballot was well, not just the barcode.

The barcode consists of several things:

- Identifiers, such as the date (but *not* time), election, precinct, type of ballot, polling machine, and random ballot ID for reconciliation against the electronic record made by the Electronic Voting Station or the Electronic Voting Station with Reading Impaired Interface. No information that can identify the voter is included on the ballot.
- The selections made by the voter.
- Checksums to detect processing errors.
- Additional padding data to obscure the barcode so that poll workers, who will be able to see the barcode (but not the textual part of the ballot) will not be readily able to ascertain by eye what selections the voter made.
- The barcode is designed so that none of the information in the barcode can be used to identify any voter personally.

Spoiled paper ballots are kept by the Ballot Reconciliation System to be reconciled against Electronic Ballot Images (EBIs) produced by the Electronic Voting Station or the Electronic Voting Station with Reading Impaired Interface.

5.1.5 Privacy Folder

The paper ballot contains the voter's choices in two forms: a form that can be read by people and a barcode that expresses those choices in a machine-readable form.

Poll workers may come in contact with the ballot should they be asked to assist a voter or to cast the ballot into the ballot box. In order to protect voter privacy it is desirable to minimize the chance that a poll worker might observe the voter's ballot choices.

A privacy folder is just a standard file folder with an edge trimmed back so that it reveals only the barcode part of a ballot. The voter is expected to take his/her ballot from the printer of the Electronic Voting Station or the Electronic Voting Station with Reading Impaired Interface and place it into a privacy folder before leaving the voting booth.

The privacy folder is designed so that the voter may place the ballot still in its folder against the scanning station of Ballot Verification Station to hear the voter's ballot's choices spoken.

When handed the ballot by the voter, the poll worker casts the ballot by turning the privacy folder so the ballot is face down, and then sliding the paper ballot into the ballot box.

5.1.6 Ballot Box

This is a physically secure container, into which voters have their paper ballots placed, in order to "cast" their votes. The mechanical aspects of the voting box will vary from jurisdiction to jurisdiction, depending on local laws and customs.

5.1.7 Ballot Verification Station

The Ballot Verification Station reads the ballot produced by the Electronic Voting Station or the Electronic Voting Station with Reading Impaired Interface and speaks (auditorily) the selections on the voter's ballot. A count is kept of usage, including counts of consecutive usage for the same ballot, but no permanent record is kept of which ballots are verified.

The computer boots off the CD-R, which includes the following:

- The operating system
- The BVS software
- Ballot Definition files and public keys of various Electronic Voting Stations
- Sound files for the ballot
- Personalization
- Startup record
- Non-ballot identifying statistics on usage

It is possible for the Ballot Verification Station to have a screen and to display the selections on the screen at the voter's option. Such an option (enabled by the voter upon her request) would enable a voter who can read to verify that her ballot will be read correctly for automated tallying.

The Ballot Verification Station reads the same portion of the paper ballot read by the Ballot Reconciliation Station, including the barcode and, ideally, the text via OCR.

5.1.8 Ballot Reconciliation Station

The Ballot Reconciliation Station reads the paper ballots and reconciles them against the Electronic Ballot Images (EBIs) on the CD-Rs from the Electronic

Voting Station or the Electronic Voting Station with Reading Impaired Interface. The purpose of reconciling the paper ballots with the electronic ballot images is to prevent ballot stuffing as well as an audit check on the canvassing process. The process also produces the vote totals for that precinct.

The Ballot Reconciliation Station includes the following components:

- Scanner, preferably page fed
- PC
- Monitor
- Input devices: keyboard, mouse
- Printer
  - Prints vote totals for posting
- CD-R
  - Like the other CD-R; includes cumulative copy of EBIs as well as vote totals by precinct.
- CD drive (not writeable)
  - For loading the CD-R's from the Voting Stations.

The Ballot Reconciliation System runs the Ballot Reconciliation Procedure, which is beyond the scope of this paper.

5.1.9 Box for Spoiled Ballots

When a voter spoils a ballot, perhaps because the ballot does not accurately reflect her preferences, the ballot is marked spoiled and placed in a box for spoiled ballots for later reconciliation.

5.1.10 Box for Provisional Ballots

When a voter shows up at a polling place and does not appear on the voting roll or the voting roll shows that the voter was sent an absentee ballot, then the voter is allowed to vote by being given a "token" for a provisional ballot. A distinctive smart card or a provisional "blank" ballot or even a distinctive privacy folder is given to the voter. When the voter has printed the provisional ballot and hands it to the poll worker, the poll worker seals the provisional ballot in an envelope along with the details necessary to determine whether the ballot should be counted and places the provisional ballot in a box for provisional ballots for later reconciliation.

5.2 Absentee Ballots and Manual Polling-Place Ballots

Paper optical-scan ballots will be used for absentee ballots and also for the manually cast polling-place ballots.

5.2.1 Format and Marking

The format and marking of the absentee ballots will be similar to those of existing optical scan ballot systems.

5.2.2 Acceptance at Polling Place

When an absentee ballot is received at a polling place, a poll worker checks the identification of the person delivering it, places on the envelope a sticker from the absentee ballot audit sheet, and places it in the absentee ballot box.

Manual polling-place ballots are placed in the manual ballot box.

5.2.3 Acceptance by Mail or In-Person at County

When an absentee ballot is received by mail, a county poll worker places on the envelope a sticker from the absentee ballot audit sheet, and places it in the absentee ballot box. When an absentee ballot is hand delivered at the county canvassing site, a county poll worker checks the identification of the person delivering it, places on the envelope a sticker from the absentee ballot audit sheet, and places it in the absentee ballot box.

5.2.4 Validation

The process for validating absentee ballots is comparable to the current process, with the notable exception that the sticker must be present on the envelope. The database record for the voter needs to be marked to indicate an absentee ballot was cast. When a voter casts an absentee ballot also casts a provisional ballot, the provisional ballot will not be counted. The ballot is separated from the envelope for canvassing.

5.2.5 Canvassing

A canvassing system for absentee and provisional ballots will be developed that reads in each optically scanned ballot to create an electronic ballot image. These electronic ballot images are aggregated with the scanned or reconciled versions of the electronic voting machine-printed paper ballots.

5.2.6 Reporting

Reports are made available by precinct for the vote totals for the combination of absentee and provisional ballots. The number of absentee and of provisional ballots is also made available by precinct.

6. Current Status and Next Steps

A demonstration system was shown at the Santa Clara County Government Building in San Jose, California on April 1, 2004. This demonstration was featured on

KGO-TV and KCBS and KGO radio later that day and described in the San Jose Mercury News that morning.[49] On April 8, 2004, the San Jose Mercury News referred to our system in an editorial as a "Touch Screen Holy Grail."[50] Further demonstrations were given at the Computers, Freedom, and Privacy conference in Berkeley, California on April 23, 2004.[51] Another demonstration was given at the PlaNetwork conference in San Francisco, California on June 6, 2004.[52] The privacy aspects of our system were presented at the Workshop on Privacy in the Electronic Society[53] and will appear in a book.[54] The open source of the demonstration system can be found on Source Forge project EVM2003.[55]

Several state colleges and the Open Voting Consortium are currently in discussions with their respective Secretaries of States to obtain HAVA funding to build production-quality reference versions of this system.

7. Conclusions

The Open Voting Consortium has demonstrated a voting system based on a PC-based electronic voting machine with voter-verifiable accessible paper ballot. We have described the design for the production system we propose to build, based on the prototype we have built and the lessons learned in the process. In the development of this system, we expect to enhance the state of the art in building reliable and trustworthy computerized systems. However, it is not merely the software and hardware components that are of concern; the voting processes and procedures are also key to the development of a reliable, secure, trustworthy and accessible system.

8. Acknowledgements

We acknowledge the efforts of the volunteers of the Open Voting Consortium who contributed to the design we describe. In particular, Alan Dechert developed much of the design and Doug Jones provided significant insights into voting issues. Arthur Keller organized the development of the software and arranged for the demonstrations. The demonstration software was largely developed by Jan Kärrman, John-Paul Gignac, Anand Pillai, Eron Lloyd, David Mertz, Laird Popkin, and Fred McLain. Karl Auerbach wrote an FAQ on which parts of this paper is based. Amy Pearl also contributed to the system description. Joseph Lorenzo Hall contributed useful background. Ron Crane gave useful feedback.

Our work was inspired by Curtis Gans, Roy Saltman, Henry Brady, Ronnie Dugger, Irwin Mann, and others who have spoken out on the need for auditable, consistent, secure and open election administration. In the last two years, David Dill and Bev Harris have been especially helpful. David Dill referred several people to the OVC, and he and Bev Harris have helped the public recognize the need for a voter-verified paper audit trail.

9. References

[1] Dorian Miller, "BMW 745 Bug," September 22, 2002, found at http://www.cs.unc.edu/~dorianm/academics/comp290test/bmw745bug.html

[2] Greg Clark, Staff Writer and Alex Canizares, "Navigation Team Was Unfamiliar with Mars Climate Orbiter," posted November 10, 1999, found at http://www.space.com/news/mco_report-b_991110.html

[3] Ken Thompson, "Reflections on Trusting Trust," *Communication of the ACM*, Vol. 27, No. 8, August 1984, pp. 761-763, found online at http://www.acm.org/classics/sep95/ .

[4] In contrast, we propose to use a variant of Linux called Knoppix that can boot directly off a CD.

[5] See http://www.wired.com/news/evote/0,2645,65173,00.html

[6] See http://www.wired.com/news/evote/0,2645,65031,00.html

[7] The Help America Vote Act of 2002 (HAVA), 42 U.S.C.A. §§ 15301 - 15545 (West 2004). See http://fecweb1.fec.gov/hava/hava.htm

[8] Lorrie Faith Cranor, "Voting After Florida: No Easy Answers," March 19, 2001, available from http://lorrie.cranor.org/voting/essay.html

[9] Federal Election Commission, Voting System Standards, Vols. 1 & 2 (2002), available at http://www.fec.gov/pages/vssfinal/ (Microsoft Word DOC format) or http://sims.berkeley.edu/~jhall/fec_vss_2002_pdf/ (Adobe PDF format).

[10] We propose that each polling place include a Ballot Reconciliation System (BRS) (described below) that has a sheet fed scanner for reading the barcode. The BRS has other benefits in support of auditing the system, also described below.

[11] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach, *Analysis of an Electronic Voting System*, Proc. IEEE Symposium on Security and Privacy (May, 2004), found at http://avirubin.com/vote/analysis/index.html

[12] *Risk Assessment Report: Diebold Accuvote-TS Voting System and Processes (redacted)*, Science Applications International Corporation SAIC-6099-2003-261, Sept. 2, 2003. See: http://www.dbm.maryland.gov/SBE

[13] Secretary of State Kevin Shelley Announces Directives To Ensure Voter Confidence in Electronic Systems, Nov. 21, 2003. See http://www.ss.ca.gov/elections/ks_dre_papers/ks_ts_press_release.pdf

14 "E-Voting Undermined by Sloppiness," Wired News, December 17, 2003. See http://www.wired.com/news/evote/0,2645,61637,00.html?tw=wn_tophead_2
15 Secretary of State Kevin Shelley Bans Diebold TSx for Use in November 2004 General Election, April 30, 2004. See http://www.ss.ca.gov/executive/press_releases/2004/04_030.pdf
16 "Leahy: Unskilled workers to blame," Miami Herald, September 12, 2002.
17 "Area Democrats say early votes miscounted," The Dallas Morning News, October 22, 2002.
18 "Electronic Ballots Fail to Win Over Wake Voters, Election Officials, Machines Provide Improper Vote Count at Two Locations," WRAL-TV Raleigh-Durham, November 2, 2002.
19 "ESlate voting proves smooth, not flawless," Houston Chronicle, November 5, 2003.
20 "Polling places report light turnout here," Richmond Times-Dispatch, February 11, 2004.
21 "Voters Decide Record Bond Issue; Edwards Quits," NBC4TV, March 2, 2004.
22 "7,000 Orange County Voters Were Given Bad Ballots." Los Angeles Times, March 8, 2004.
23 "Human goofs, not machines, drag vote tally into next day." Palm Beach Post, March 14, 2002.
24 "Out of Touch: You press the screen. The machine tells you that your vote has been counted. But how can you be sure?" New Times, April 24, 2003.
25 "Officials still searching for election glitch: The new system could not send the tabulations to the elections office." St. Petersburg Times, April 6, 2002.
26 "Elections Chief Sees Nearly Flawless Vote." St. Petersburg Times, March 5, 2002.
27 "Election results certified after software blamed." Albuquerque Tribune, November 19, 2002.
28 "Montville and Chatham mayors ousted." New Jersey Star-Ledger, June 9, 2004.
29 See http://www.wired.com/news/business/0,1367,58738,00.html
30 See http://www.accupoll.com/News/PressReleases/2003-10-10.html
31 See http://www.sequoiavote.com/mediadetail.php?id=74
32 See http://www.wired.com/news/evote/0,2645,63618-2,00.html
33 See http://www.siliconvalley.com/mld/siliconvalley/9647591.htm
34 See http://www.aitechnology.com/votetrakker2/evc308.html
35 These systems are provided by ES&S (http://www.essvote.com/HTML/products/automark.html) and Vogue Election Products and Services (http://www.vogueelection.com/products_automark.html).
36 See http://varbusiness.com/article/showArticle.jhtml?articleId=18841335
37 See http://www.populex.com/
38 See http://www.election.nl/
39 Margaret Anne McGaley. *Electronic Voting: A Safety Critical System.* See http://www.redbrick.dcu.ie/~afrodite/E-Voting/Report/node18.html
40 Election Commission of India, Electronic Voting Machine (EVM) description. See http://eci.gov.in/EVM/
41 See http://www.smartmatic.com/electionsVenezuela2004.htm
42 See http://www.cartercenter.org/viewdoc.asp?docID=2023&submenu=news and http://www.cartercenter.org/documents/2020.pdf (page 22).
43 See http://www.verifiedvoting.org/
44 The OVC proposes to audit *every* vote by comparing each paper ballot with its electronic copy.
45 See http://www.accessiblesociety.org/topics/voting/electionreformlegis.html
46 See http://www.verifiedvoting.org/
47 See http://www.ss.ca.gov/elections/ks_dre_papers/avvpat_standards_6_15_04.pdf and http://www.ss.ca.gov/elections/ks_dre_papers/press_release_avvpat_06_15_04.pdf
48 Each ballot's barcode will be digitally signed. We use the private key for signing each ballot and the public key verifying the signature. The key pair could be generated by the voting machine, with the public key recorded on the CD-R and on the "test sheet" printed at the start of a voting day and the private key never released outside the voting machine itself.
49 See http://www.siliconvalley.com/mld/siliconvalley/8328014.htm
50 See http://www.kentucky.com/mld/mercurynews/news/opinion/8383100.htm
51 See http://cfp2004.org/program/#votingmachinedemo
52 See http://www.planetwork.net/2004conf/program.html
53 Arthur M. Keller, David Mertz, Joseph Lorenzo Hall, and Arnold Urken, "Privacy Issues in an Electronic Voting Machine," (Extended Abstract) in *Workshop on Privacy in the Electronic Society WPES 2004*, October 28, 2004, Washington DC.
54 Arthur M. Keller, David Mertz, Joseph Lorenzo Hall, and Arnold Urken, "Privacy Issues in an Electronic Voting Machine," in *Privacy and Identity: The Promise and Perils of a Technological Age*. Katherine J. Strandburg and Daniela Raicu, eds., Kluwer Academic Publishing, to appear.
55 See http://evm2003.sourceforge.net/

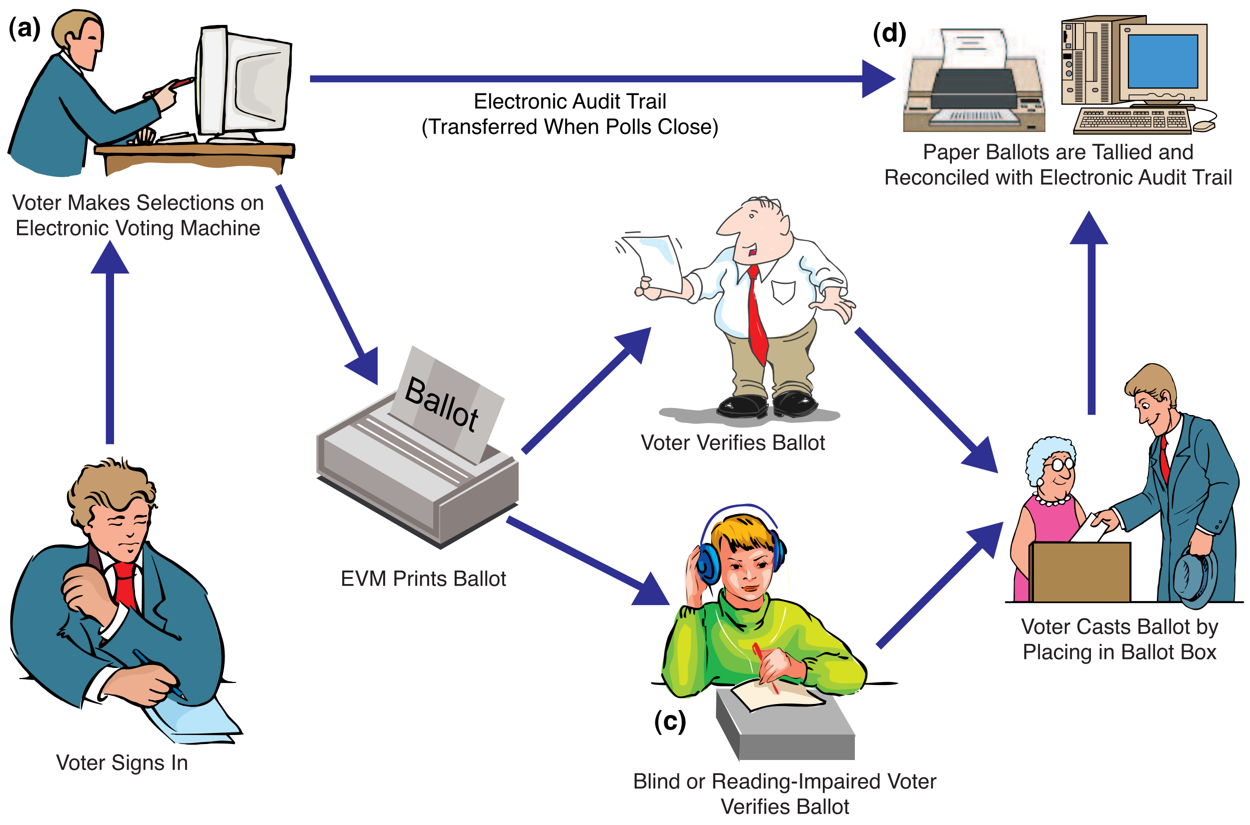

(a)
Voter Makes Selections on Electronic Voting Machine
Electronic Audit Trail (Transferred When Polls Close)
(d)
Paper Ballots are Tallied and Reconciled with Electronic Audit Trail
Voter Verifies Ballot
Ballot
EVM Prints Ballot
Voter Casts Ballot by Placing in Ballot Box
Voter Signs In
(c)
Blind or Reading-Impaired Voter Verifies Ballot

Figure 1. Schematic of Precinct/Polling Place Element.
(a) Electronic Voting Station
(b) Electronic Voting Station with Reading-Impaired Interface (not shown)
(c) Ballot Verification Station
(d) Ballot Reconciliation Station

Auto-pilot: A Platform for System Software Benchmarking

Charles P. Wright, Nikolai Joukov, Devaki Kulkarni, Yevgeniy Miretskiy, and Erez Zadok
Stony Brook University

Abstract

When developing software, it is essential to evaluate its performance and stability, making benchmarking an essential and significant part of the software development cycle. Benchmarking is also used to show that a system is useful or provide insight into how systems behave. However, benchmarking is a tedious task that few enjoy, but every programmer or systems researcher must do. Developers need an easy-to-use system for collecting and analyzing benchmark results. We introduce *Auto-pilot*, a tool for producing accurate and informative benchmark results. Auto-pilot provides an infrastructure for running tests, sample test scripts, and analysis tools. Auto-pilot is not just another metric or benchmark: it is a system for automating the repetitive tasks of running, measuring, and analyzing the results of arbitrary programs. Auto-pilot can run a given test until results stabilize, automatically highlight outlying results, and automatically detect memory leaks. We have used Auto-pilot for over three years on eighteen distinct projects and have found it to be an invaluable tool that saved us significant effort.

1 Introduction

Benchmarking contributes evidence to the value of work, lends insight into the behavior of systems, and provides a mechanism for stress-testing software. However, benchmarking can be an arduous task. Benchmarking takes a lot of time, and the initial iteration of benchmarks often exposes bugs or inefficient code. After changes are made to the code, the benchmarks need to be repeated, time and time again. Once the code is bug free and stable, there are sometimes unexpected results. The output then needs to be examined to determine a cause. If no cause for the suspicious results is found, then the benchmarks need to be repeated without changing any parameters to verify the unexpected results. After the results are verified, one variable needs to be changed at a time to narrow down the source of the erroneous data. This cycle of benchmarking and analysis is often repeated many times.

We have identified two primary considerations when collecting and analyzing benchmark results:

Accuracy The numbers that benchmarks produce need to be mathematically correct, but an even more difficult requirement to satisfy is that they need to be reproducible, stable, and fair. They should be reproducible so that you can re-run the test and get similar results. This way if you need to make slight modifications, it is possible to go back and compare results. To help achieve this, Auto-pilot records pertinent system information (e.g., the OS version, hard disk model, and partitions) so that you can reproduce the test environment as closely as possible. Tests also need to be reproducible so that others can verify your results. For example, each test should be run under similar circumstances (e.g., a cold or warm cache). If one test fails, then it may impact all future tests, so a series of tests should be stopped so that erroneous results are not included.

Presentation The accuracy of results is inconsequential if the results can not be understood and correctly interpreted. Each benchmark usually results in hundreds or thousands of numbers. For example, each execution of a program results in elapsed, user, and system time. Additionally, the CPU utilization and wait (I/O) time are reported. If there is a four-threaded benchmark that is run twenty times, then 400 values are produced. There are also usually many configurations of a benchmark. Each variable that is introduced multiplicatively increases this value. In this example, if we ran the test for 1, 2, 4, 8, 16 and 32 threads, there would be a total of 6,300 values. For every additional quantity that is measured, another 1,260 numbers are generated. Without assistance it is rather tedious to wade through this sea of numbers.

Auto-pilot is an infrastructure that produces accurate and informative results. Auto-pilot is not a metric or a specific benchmark, but rather a framework to run benchmarks. Auto-pilot includes a language for describing a series of benchmarks using a simple syntax that includes basic loops and conditionals. The Auto-pilot distribution includes a set of scripts for several compile benchmarks and also for running multiple concurrent Postmark processes [12]. Auto-pilot also provides useful analysis tools that handle multi-process benchmarks that can also account for background processes or kernel threads. Auto-pilot can automatically stop the benchmarks after reliable results are obtained, highlight outlying values, and detect memory leaks. Results are presented in a tabular format that can easily be imported into spreadsheets. Auto-pilot also includes a bar and line graph script that generates graphs from the tabular results using Gnuplot.

One of the most frustrating tasks when benchmarking is deciding how many times to run a test. The ideal situation is to run the benchmark precisely as many times as needed to obtain stable results, but not any more than that (to save time for both the benchmarker and to free the testbed for more tests). Auto-pilot can automatically determine when test results have met an arbitrary stability condition (e.g., the half-width of the confidence interval is within 5% of the mean) and stop the tests at that point.

Benchmarking is often used to debug software. Software may leave some state behind that affects future results. For example, a kernel module may have small memory leaks that eventually exhaust system resources. Auto-pilot uses linear regression to determine if there are memory leaks (which reduce the total amount of free memory) , or a performance degradation after each iteration of the benchmark.

Even software that works correctly changes the state of the system. For example, a working set of files is loaded into OS caches. Often, researchers unmount and remount the file system to provide cold cache results, but rebooting has measurable advantages over simply remounting the file system. Unfortunately, rebooting usually adds time-consuming manual intervention to the benchmarking process. To ensure a consistent system state, Auto-pilot provides checkpointing support. Checkpointing and rebooting is fully automated: all Auto-pilot state is written to a file, the machine restarts, and after system initialization completes Auto-pilot resumes the benchmarks from where it left off.

Systems have a long life cycle and benchmark results may need to be analyzed months or years after they were taken (e.g., after a paper is accepted, reviewers often ask questions about the performance evaluation). Complete results must be saved for future analysis in a meaningful way. Auto-pilot carefully stores relevant system information and all program output.

The rest of this paper is organized as follows. We describe previous work in Section 2. In Section 3 we describe the design and implementation of Auto-pilot. We conclude in Section 4.

2 Background

There are many metrics and benchmarks available to test systems. In this section we describe a few notable systems and how they differ from Auto-pilot.

Several tools seek to measure precisely (down to microseconds, nanoseconds, or CPU cycles) the amount of time it takes to perform a single operation. Two examples of these tools are lmbench [14] and hbench:OS [5]. These tools include both a set of tests, and also analysis tools for these specific tests. For example, lmbench measures latency and bandwidth for memory, IPC, file system operations, disk I/O, cached I/O, TCP, UDP, and RPC. Several O/S primitives are also measured by lmbench, including system call entry, context switching, signal handling, process creation, and program execution (`fork+exec`). These precise measurements are useful to debug parts of a system, but they do not measure the interactions between system calls that exist in more realistic workloads. The lmbench suite contains not only the infrastructure to run and report these benchmarks, but defines the tests as well. In contrast, for Auto-pilot we focus on running relatively large scale tests (whole programs), like Postmark, a compile, or micro-benchmarks that perform some operation many times. For example, we have used Auto-pilot to benchmark specific file system operations like `stat` and `readdir` by running `find` over a tree of files.

Brown and Seltzer developed hbench:OS, which is a modified version of lmbench that improves timing and statistical methodology, adds more parameters to tests, and improves individual benchmarks. In lmbench many tests are run in a loop and a final result is calculated based on all the runs, but some tests are only run once. Because some architectures have coarse-grained timing infrastructures, running the test may produce inaccurate results (e.g., 0 microseconds to perform an operation like a TCP connect). To remove this deficiency, hbench:OS uses a self-scaling loop that runs the test for at least one second, which is several orders of magnitude more than even the worst timing mechanisms. For tests that can be run only once, hbench:OS uses CPU cycle counters. Different methods are used to report results in lmbench and hbench:OS. For some lmbench tests a mean is reported, for others a minimum. In hbench:OS each individual measurement is recorded so that data analysis is separated from reporting. In hbench:OS, $n\%$ trimmed means are used for all results. The lowest and highest $n\%$ results are discarded, and the remaining $(100 - 2n\%)$ results are used to compute an arithmetic mean. Whether trimmed means are a better method for results analysis is disputed by the lmbench authors, but because hbench:OS stores raw results, different types of analysis are still possible. Overall, hbench:OS and Auto-pilot are different for the same reasons lmbench and Auto-pilot are different: we focus on large general purpose benchmarks, whereas these two systems focus on small micro-benchmarks. However, in Auto-pilot we have made some similar decisions to hbench:OS for reporting and analysis. In Auto-pilot, we record all results and test output. Auto-pilot is also flexible with support for arbitrary metrics (e.g., elapsed time, I/O operations, or packets sent). Our analysis tools can then operate on the raw output to produce reports and graphs. Like hbench:OS, Auto-pilot can also automatically scale benchmarks to the testbed (e.g., run a benchmark for one hour or until the confidence interval's half-width meets a threshold).

Profilers and other tools measure how long specific sections of code are executed (e.g., functions, blocks, lines or instructions) [3, 9]. Profilers can be useful because they often tell you how to make your program faster, but the profiling itself often changes test conditions and adds overhead, making it unsuitable for comparing the performance of two systems. The Perl Benchmark::Timer module [10] times sections of Perl code within a script. Before the code section, a start function is called; and after the section an end function is called. Benchmark::Timer can run a code section for a fixed number of times, or alternatively until the width of a confidence interval meets some specified value. Auto-pilot can use similar methodology to determine how many iterations of a test should be run.

Other benchmarks like SDET [7] and AIM7 [1] are system-level benchmarks. Both SDET and AIM7 run a pre-configured workload with increasing levels of concurrency. The metric for each benchmark is the peak throughput. These systems focus on developing a metric, and measuring that specific metric, not running arbitrary benchmarks.

The closest system to Auto-pilot is Software Testing Automation Framework (STAF) [11] developed by IBM. STAF is an environment to run specified test cases on a peer-to-peer network of machines. Rather than measuring the performance of a given test, STAF aims to validate that the test case behaved as expected. STAF runs as a service on a network of machines. Each machine has a configuration file that describes what services the other machines may request it to perform (e.g., execute a specific program). STAF also provides GUI monitoring tools for tests. The major disadvantage with STAF is that it requires complex setup, does not focus on performance measurement, and is a heavy-weight solution for running multiple benchmarks on a single machine

The Open Source Development Lab (OSDL), provides a framework called the Scalable Test Platform (STP) that allows developers to test software patches on systems with 1–8 processors [13]. STP allows developers to submit a patch for testing, and then automatically deploys the patch on a system, executes the test, and posts the results on a Web page. STP makes it relatively simple to add benchmarks to the framework, but the benchmarks themselves need to be changed to operate within the STP environment. Auto-pilot is different from STP in two important ways. First, STP is designed for many users to share a pool of machines, whereas Auto-pilot is designed to repeatedly run a single researcher's set of tests on a specific machine. Second, STP provides no analysis tools; it simply runs tests and logs the output, whereas Auto-pilot measures processes and provides tools to analyze results.

3 Design

To run a benchmark, you must write a configuration file that describes which tests to run and how many times. The configuration file does not describe the benchmark itself, but rather points at another executable. This executable is usually a small wrapper shell script that provides arguments to a program like Postmark or a compile benchmark. The wrapper script is also responsible for measurement. We provide sample configuration files and shell scripts for benchmarking file systems. These can be run directly for common file systems, or easily adapted for other types of tests. Given a configuration file and shell scripts, the next step is to run the configuration file with Auto-pilot. Auto-pilot parses the configuration file and runs the tests, producing two types of logs. The first type is simply the output from the programs. This can be used to verify that benchmarks executed correctly and to investigate any anomalies. The second log file is a more structured results file that contains a snapshot of the system and the measurements that were collected. The results file is then passed through our analysis program, *Getstats*, to create a tabular report. Optionally, the tabular report can be used to generate a bar or line graph.

In Section 3.1 we describe `auto-pilot`, the Perl script that runs the benchmarks and logs the results. In Section 3.2 we describe the sample shell scripts specific to the software being benchmarked. In Section 3.3 we describe Getstats, which produces summaries and statistical reports of the Auto-pilot output. In Section 3.4 we describe our plotting scripts. In Section 3.5 we describe and evaluate checkpointing and resuming benchmarks across reboots. In Section 3.6 we describe using hooks within the benchmarking scripts to benchmark NFS.

3.1 `auto-pilot`

The core of the Auto-pilot system is a Perl script that parses and executes the benchmark scripts. Each line of a benchmark script contains a command (blank lines and comments are ignored). The command interface has been implemented to resemble the structure of a typical programming language. This makes Auto-pilot easy and intuitive to enhance when writing new benchmarks. Next, we describe the thirteen primary commands.

TEST begins a benchmark. The test directive takes between two and four arguments. The first argument is the name of the test, which is also used to build the name of the output file. The second argument is the minimum number of times to run the test. The third and fourth arguments are optional. The fourth argument specifies a program that determines if the test should continue. If the program's exit status is zero, then the benchmark stops otherwise it continues. The third argument is how many iterations of the benchmark to execute before rerunning the program specified in the fourth argument. This argument is useful because it may be more efficient to execute several iterations of the benchmark between runs of the program rather than running a potentially computationally intensive program after a single iteration. Using a program to determine if tests should continue allows the benchmark to execute until there are stable results, thereby saving time on the testbed, and leaving more time for analysis. Getstats, described in Section 3.3, has support for arbitrary predicates to determine stability. The test directive contains SETUP, EXEC, and CLEANUP directives (described below). Other control directives are also allowed (e.g., IF directives, variable declarations, and loops).

THREADS tells Auto-pilot how many concurrent benchmark processes should be run at once. This can be used to test the scalability of systems. Our analysis tools, described in Section 3.3, aggregate the results from these threads.

EXEC executes a test. If the THREADS directive is used, then $threads$ processes are simultaneously executed. The results for each thread are logged to separate files. Auto-pilot sets the environment variable `APTHREAD` to 1 in the first thread, 2 in the second thread, and N in the n-th thread. This allows each thread to perform slightly different tasks (e.g., performing tests in different directories).

SETUP executes a setup script for a test. Setup scripts are not multi-threaded, and are used for initialization that is common to all threads (e.g., to mount a test file system). We also support a PRESETUP directive that is run only once. One possible use of PRESETUP is to format a file system only once, and remount it before each test using a SETUP directive.

CLEANUP scripts are used to undo what is done in a setup script. CLEANUP scripts can be used to ensure a cold cache for the next test. To run a script after all the iterations of the test are completed, a POSTCLEANUP directive can be used for final cleanup.

VAR sets an Auto-pilot variable. Simple variable substitution is performed before executing each line: `%VAR%` is replaced with the value of `VAR`. These variables are not exposed to external processes.

ENV sets an environment variable and the corresponding Auto-pilot variable. This can be used to communicate with benchmark scripts without the need for many command-line arguments. Auto-pilot also replaces `$VAR$` with the contents of the environment variable `VAR`. This is similar to the `export` command within the Bourne shell.

IF is a basic conditional that supports equality, greater-than, and less-than (and their inverses). If the condition evaluates to true, then all statements until ELSE or FI are executed. If the condition is false, then the optional statements between ELSE and FI are executed. ELSE-IF blocks are also supported.

WHILE repeatedly executes a block of code while a certain condition holds true. The condition syntax WHILE uses is the same as IF.

FOREACH assigns multiple values to one variable in turn. This is useful because often a single test needs to be repeated with several different configurations.

FOR is similar to FOREACH, but instead of specifying each value explicitly, a start, end, increment, and factor are specified. For example, FOR THREADCOUNT=1 TO 32 FACTOR 2 would execute the loop with a THREADCOUNT of 1, 2, 4, 8, 16, and 32.

FASTFAIL causes Auto-pilot to abort if one of the benchmarks may not be successful. If the benchmarks continue, then they may destroy important state that could lend insight into the cause of the failure. An optional *fastfail* script is also defined, which can be used to send email to the person responsible for the benchmarks. We have found it very useful to email pagers, so that testbeds do not remain idle after a failed benchmark.

CHECKPOINT writes *all* Auto-pilot internal state to a file. If Auto-pilot is invoked with the checkpoint file as an argument, then it resumes execution from where it left off. The return value of CHECKPOINT is similar to the Unix `fork` system call. After restoring a checkpoint, the value of the Auto-pilot variable RESTORE is 1, but after writing the checkpoint, the value of RESTORE is 0. In Section 3.5 we present and evaluate an in-depth example of checkpointing across reboots.

Configuration example Figure 1 shows the Postmark configuration file included with Auto-pilot.

```
1  INCLUDE common.inc
2  FOREACH FS ext2 ext3 reiserfs
3    FOR THREADCOUNT=1 TO 32 FACTOR 2
4      THREADS=%THREADCOUNT%
5      TEST %FS%:%THREADS% 10 1 getstats \
  --predicate '("$delta" < 0.05 * $mean) \
  || ($count > 30)'
6        SETUP fs-setup.sh %FS%
7        EXEC postmark.sh
8        CLEANUP fs-cleanup.sh %FS%
9      DONE
10   DONE
11 DONE
12 INCLUDE ok.inc
```

Figure 1: A sample Auto-pilot configuration file

Line 1 includes `common.inc`, a configuration file that performs actions that are common to all tests. `common.inc` also turns off all excess services (e.g., Cron) to prevent them from interfering with a benchmark run. The `common.inc` file also includes `local.inc`, which the user can create to set variables including `TESTROOT`, which defines the directory where the test will take place. Line 2 begins a `FOREACH` loop. This loop is executed a total of three times. The first time the value of `FS` is set to “ext2,” the second time it is set to “ext3,” and on the third it is “reiserfs.” Line 3 begins a nested loop that will set the `THREADCOUNT` variable to 1, 2, 4, 8, 16, and 32. Line 4 sets the number of threads to use to `THREADCOUNT`. Lines 5–9 define the test. The test is named `%FS%:%THREADS%`. For example, the first test is named `ext2:1`. The test is executed 10 times, and then after each test the Getstats program (described in Section 3.3), is used to determine whether the confidence interval has an acceptable half-width for elapsed, user, and system time. The test is stopped after 30 runs, to prevent a test from running forever. Line 6 calls `fs-setup.sh` with an argument of `ext2`. This is our file system setup script, which formats and mounts a file system (formatting can be disabled with environment variables). Line 7 executes `postmark.sh`. This shell script creates a Postmark configuration, and then executes Postmark through our measurement facility (described in Section 3.2). Line 8 unmounts the Ext2 file system, so that the next run takes place with a cold cache. Lines 9–11 close their corresponding loops. Line 12 turns services back on, and optionally sends an email to the user indicating that the benchmarks are complete.

3.2 Benchmark Scripts

We provide a set of file system benchmarking scripts for Postmark [12] and compiling various packages. We included an example script for compiling Am-Utils [15], GCC [8], and OpenSSL [17]. Most other packages can be compiled by setting environment variables or with minimal changes to the existing scripts.

Many applications can be benchmarked without any scripts, but file systems require complex setup and cleanup. Our scripts also serve as an example for benchmarking applications in other domains. We wrote scripts to test some of our complex file systems. These systems required extensive testing and Auto-pilot allowed us to methodically test and debug them.

The flow of the scripts is organized as two components. The first component mounts and unmounts the file system. The second performs the Postmark or the compile benchmark. Though we distribute scripts for benchmarking compilations and Postmark on Ext2, Ext3, and Reiserfs, it is easy to add other benchmarks or platforms. To test additional file systems, the Auto-pilot scripts have hooks for `mount`, `unmount`, `mkfs`, `tunefs`, and more. With these hooks, new file systems or new file system features can be used. We provide example hooks for enabling HTrees on Ext2/3 [16] and to benchmark on top of stackable file systems [21]. We describe how we used the hooks to concurrently benchmark an NFS client and server in Section 3.6.

The sample Auto-pilot scripts we distribute demonstrate the following principles:

- Separating the test and the setup. Most benchmarks involve performing several workloads on several configurations. Each configuration has a setup and a corresponding cleanup script. Each workload has a script common to all configurations.
- Using variables for all values that may change. We additionally have support for operating system and host-specific options.
- Unmounting the file system on which the benchmark takes place between runs, even if the previous run failed.

Auto-pilot includes `common.inc`, which calls `noservices.sh` to shut down Cron, Sendmail, Anacron, LPD, Inetd, and other services on the testbed. If no user is logged in via SSH, then SSH is also turned off. We create `/etc/nologin` to prevent users from logging in while the test is being run. Swap space is optionally disabled to prevent it from affecting results. All these actions are taken to avoid unexpected user or system activity from distorting the results.

The file system setup script loads in machine-specific settings (e.g., which device and directory to use). Next, the script logs some vital statistics about the machine (OS version, CPU information, memory usage, hard disk configuration, and partition layout). The script then unmounts any previously-mounted file systems, formats the device, and mounts the new file system (e.g., Ext2, Ext3, or Reiserfs). This ensures a clean cache and consistent disk layout. Because vital machine information is recorded, the test conditions can be reproduced in the future.

Next, the benchmark is executed. The compile script unpacks, configures, and compiles the package specified. The compile commands are run through a function named *ap_measure*. This function produces a block of information that encapsulates the results for this test. By default, the block includes the iteration of the test, which thread was running, the elapsed time, the system CPU time, the user CPU time, and the command's exit status. When analyzing the results, this block logically becomes a row in a spreadsheet. Additional fields can be added through the use of measurement hooks. Measurement hooks are executed before and after the test. Each hook can produce values to be included in the block. We provide sample hooks that measure the number of I/O requests for a given partition; the amount of free memory; the amount of CPU time used by background processes; network utilization; and more. Programs such as CFS [4] or Cryptoloop [18] use a separate thread for some processing. Even if the instrumented process itself does not have additional threads, the kernel may use asynchronous helper threads to perform certain tasks (e.g., `bdflush` to flush dirty buffers or `kjournald` to manage journalized block devices). This hook allows measurement of daemons that expend effort on behalf of the measured process. The measurement hook is also called one final time so that it can produce its own blocks. This feature is used for the CPU time difference measurement hook to enumerate all processes that used additional CPU time. Our I/O measurement script also adds a block with the number of I/O operations that occurred on each partition.

The Postmark script is more complicated because it can run several Postmark processes in parallel. When running a multithreaded test, Auto-pilot sets two environment variables: `APTHREAD` and `APIPCKEY`. `APTHREAD` is the thread number for this test. `APTHREAD` can be used, for example, so that different processes use different directories. `APIPCKEY` is used for synchronization. First, the Postmark script creates a directory and then sets up a Postmark configuration file. Because Auto-pilot starts each test sequentially, one test could start before another and then the timing results would be inaccurate. To solve this, Auto-pilot creates a System V semaphore with the number of threads as its value. Before executing Postmark, each test script decrements the semaphore. We include a small C utility that calls `semctl` to decrement the semaphore. All of the test processes are suspended until the semaphore reaches zero, at which point they all begin to execute concurrently.

After all the benchmark processes complete, Auto-pilot executes the cleanup script, which unmounts the file system. After all tests are executed successfully, SSH is restarted and `/etc/nologin` is removed. This is not meant to restore the machine to its previous state, but rather to allow remote access, so that the benchmarker can retrieve results, or begin another series of benchmarks. Auto-pilot does not keep track of which services it stopped. To restore the services the machine was running before the benchmarks, it can simply be rebooted.

3.3 Getstats

Getstats is a Perl script which processes the results log file to generate useful tabular reports. In addition to Auto-pilot results files, Getstats can process Comma Separated Value (CSV) files or the output of GNU `time`. Getstats is flexible in that it does not hardcode the types of information it expects in these files; it simply reads and displays the data. The parsers themselves are also modular. Getstats searches the Perl library path for valid Getstats parsers and loads them. To write a new parser, two functions must be defined: a detection function and a parsing function that reads the input file into a two-dimensional array. We originally used this functionality to add support for GNU time files, but have also used it for some of our own custom formats. We discuss mostly timing information, but we have also analyzed network utilization, I/O operations, memory, and other quantities with Getstats.

Getstats has a basic library of functions to transform the data. Examples of transformations include adding a column derived from previous values, selecting rows based on a condition, raising warnings, grouping data based on the value of a column, or aggregating data from multiple rows to produce a single summary statistic. If Getstats detects that it is being run on results files with time data, it performs some default transformations composed of the basic library transformations. These include raising warnings if a command failed, aggregating multiple threads into a single value, computing a *wait* time (the time the measured process was not running) and a CPU-utilization column, raising warnings if any test had a high z-score for one if its values, computing overheads, and finally generating a tabular report.

Figure 2 shows a tabular report generated by Getstats compares Ext2 with Ext3, when run on the Postmark configuration discussed in Section 3.1. The first line of the report is a high z-score warning for the third iteration of ext2 with one thread. If there are tests with very large z-scores, then there likely were problems with the benchmark. A few other high z-score warnings were issued, but are not shown to conserve space. Of note is that Ext2 ran only ten times, but Ext3 needed to run 15 times to get an acceptable half-width percentage for the measured quantities (the predicates are not run for Wait

```
ext2:1.res: High z-score of 2.21853230276562 for elapsed in epoch 3.
...
ext2:1.res
NAME     COUNT   MEAN MEDIAN    LOW   HIGH    MIN    MAX SDEV%   HW%
Elapsed     10  6.055  6.063  5.991  6.120  5.855  6.180 1.491 1.067
System      10  2.758  2.760  2.709  2.807  2.640  2.880 2.499 1.788
User        10  1.675  1.680  1.615  1.735  1.510  1.820 5.044 3.609
Wait        10  1.622  1.636  1.567  1.677  1.465  1.718 4.759 3.404
CPU%        10 73.221 73.079 72.572 73.871 72.007 74.981 1.240 0.887

ext3:1.res
NAME     COUNT   MEAN MEDIAN    LOW   HIGH    MIN    MAX SDEV%   HW%      O/H
Elapsed     15 77.861 76.865 74.156 81.567 64.308 88.209 8.594 4.759 1185.869
System      15  4.272  4.290  4.217  4.327  4.100  4.410 2.334 1.293   54.895
User        15  1.825  1.820  1.773  1.877  1.670  1.990 5.132 2.842    8.935
Wait        15 71.765 70.775 68.064 75.466 58.158 82.179 9.312 5.157 4324.025
CPU%        15  7.885  7.923  7.499  8.272  6.836  9.563 8.850 4.901  -89.231
```

Figure 2: The Getstats tabular report format showing the results for Postmark with a single thread on Ext2 and Ext3.

and CPU% which are computed quantities). This is not unexpected because journaling adds complexity to the file system's I/O pattern, resulting in greater variability [6]. The tabular format itself is useful as a mechanism to present detailed results or to import data into spreadsheets, but graphs present a better overall picture of the results.

Output Modes Getstats provides several useful options to analyze and view different output modes. The simplest mode outputs the raw uninterpreted values, which is useful when there are problems with the benchmark (e.g., one run had anomalous results). The tabular report also has information useful for plotting. The LOW and HIGH columns are suitable for creating error bars with Gnuplot. Getstats supports several methods of creating error bars: Student-t confidence intervals, the minimum and maximum value, or the standard deviation. By default Getstats reports the count, mean, median, minimum, maximum, and Student-t confidence interval error bar values (shown as low and high), and the standard deviation and half-width of the confidence interval as a percent of the mean. The standard deviation is a measure of how much variance there is in the tests. The half-width of the confidence interval describes how far the true value may be from the mean with a given degree of confidence (by default 95%). If multiple results files are specified, then the first file is used as a baseline to compute overheads for the subsequent files (this is shown as "O/H"). This report gives a high-level yet concise overview of a test.

Statistical Tests After changing your software you would like to know if your changes actually had a measurable effect on performance (or some other measured quantity). In some cases it is sufficient to compare the means, and if they are "close", then you may assume that they are the same or that your change did not noticeably affect performance. In other circumstances, however, a more rigorous approach should be used. For example, if two tests are very close, it can be difficult to determine if there is indeed a difference, or how large that difference really is. To compare two samples, Getstats can compute the confidence interval for the difference between the means, and can also run a two-sample t-test. The confidence interval quickly tells you if there is a difference, and how much it is. The confidence interval is simply a range of numbers. If that range includes zero, then the samples are not significantly different. Getstats also can run a two-sample t-test to determine the relationship between two results files. A statistical test has a *null hypothesis*, which is assumed to be true. An example of a null hypothesis is $u1 = u2$. The result of a t-test is a P-value, which is the probability that you would observe the data if the null hypothesis is true. If the P-value is large (close to 1), then your data is consistent with the null hypothesis. If the P-value is small (closer to 0), then your data is not consistent with the null hypothesis. If the P-value is smaller than a predetermined significance level (e.g., .05), then you reject the original assumption (i.e., the null hypothesis).

Figure 3 shows the output of a Getstats t-test for two samples: CHILL and REMOUNT. We ran a recursive `grep` benchmark over the GCC 3.4.3 source tree. For CHILL we ran a program that we wrote called *chill* that is designed to ensure cold-cache results. Our version of chill was inspired by a similar program in SunOS 4 [20]. Chill allocates and dirties as much memory as possible, thereby forcing the kernel to evict unused objects. We hypothesized that chill would provides more stable results than simply unmounting the test file system, because it causes all caches to be purged, not just those related to the test file system. The REMOUNT configuration unmounts the file system and then mounts it again instead of running chill.

```
chill
NAME    COUNT MEAN   MEDIAN LOW    HIGH   MIN    MAX    SDEV% HW%
Elapsed 10    38.649 38.193 37.950 39.348 37.673 40.379 2.528 1.808
System  10    1.663  1.675  1.603  1.723  1.540  1.770  5.071 3.628

...
remount
NAME    COUNT MEAN   MEDIAN LOW    HIGH   MIN    MAX    SDEV% HW%   O/H
Elapsed 10    38.751 38.699 38.580 38.921 38.465 39.307 0.614 0.439 0.262
System  10    1.796  1.790  1.677  1.915  1.580  2.080  9.255 6.620 7.998
...
Comparing remount (Sample 1) to chill (Sample 2).
Elapsed: 95%CI for remount - chill = (-0.567, 0.769)
Null Hyp. Alt. Hyp. P-value Result
u1 <= u2  u1 >  u2  0.377   ACCEPT H_0
u1 >= u2  u1 <  u2  0.623   ACCEPT H_0
u1 == u2  u1 != u2  0.754   ACCEPT H_0

System: 95%CI for remount - chill = (0.009, 0.257)
Null Hyp. Alt. Hyp. P-value Result
u1 <= u2  u1 >  u2  0.018   REJECT H_0
u1 >= u2  u1 <  u2  0.982   ACCEPT H_0
u1 == u2  u1 != u2  0.037   REJECT H_0

...
```

Figure 3: The Getstats t-test output. User time, Wait time, and CPU usage are omitted for brevity.

From the tabular report, we can see that elapsed time differs by only 0.262%, and system time differs by only 7.998%. After the tabular report, each measured value in the samples is compared (e.g, elapsed time from the first sample is compared to elapsed time from the second sample).

If we examine the confidence interval for the elapsed time, we can see that it includes zero (the beginning of the range is less than zero, but the end of the range is greater than zero), which means that CHILL and REMOUNT are not distinguishable. The next four lines show the results of the t-test for elapsed time. Getstats runs the t-test with three distinct assumptions: $u_1 \leq u2$, $u_1 \geq u_2$, and $u_1 = u_2$. For simplicity, we chose to display all of the tests rather than forcing the user to specify the tests of interest. It is up to the benchmarker to determine which of these assumptions makes sense. In all cases, the P-value was higher than the significance level of (0.05), which means that we can not reject the assumptions with 95% confidence. Therefore, we conclude that CHILL and REMOUNT are indistinguishable for elapsed time.

For system time, the confidence interval does not contain zero, which means the two tests have a significant difference. This is supported by the P-value, which shows us that there is only a 3.7% chance of observing this data if the two samples were in fact equal. There is also only a 1.8% chance that CHILL uses less system time than REMOUNT. Because 3.7% and 1.8% are less than the significance level, we can reject the null hypothesis that $u_1 = u_2$ and that $u_1 \leq u_2$ where u_1 represents REMOUNT and u_2 represents CHILL. Because our assumption that REMOUNT used less system time than CHILL is false, we know that REMOUNT did use more system time because it needed to recreate objects that the kernel evicted.

Detecting Anomalies Often, one or two bad runs are not noticeable from summary statistics alone. Getstats provides two mechanisms for finding anomalous results:

- Automatically highlighting outlying values
- Performing linear regression on the values

To automatically highlight outlying values, Getstats uses the z-score of each point. The z-score is the difference between the value and the mean, divided by the standard deviation. If the z-score is greater than a configurable value (by default 2), a warning is printed before the summary statistics. The benchmarker can then look into the results further to analyze the cause of the outliers.

Getstats can also compute a least-squares linear regression to fit the elapsed time, system time, user time, and free memory. Linear regression computes the slope and intercept of the line that best fits a set of points. If the results are stable, then the slope of these should be zero. The intercept is just the magnitude of the values, so it is unimportant. If the magnitude of the slope is not close to zero, then a warning is issued. If the free memory slope is negative, then it indicates that on each iteration the software is leaking memory. If the elapsed or CPU times have positive slopes, it similarly indicates that some resource is becoming more scarce, and causing a gradual performance degradation. Depending on the system being analyzed, different thresholds are appropriate, so the default value can be overridden by the benchmarker. To ensure that the amount of free memory reported is accurate, we run our version of chill. Chill allocates as much memory as possible, then dirties every page. Chill is terminated by the Linux out-of-memory manager (or when `malloc` can no longer get additional memory). At this point the kernel has evicted objects and pages to make room for chill, and the amount of memory consumed should be at a minimum.

Predicate Evaluation Getstats evaluates predicates for use with `auto-pilot`'s `TEST` directive. Getstats runs the predicate over each column in the results file, and returns zero if the predicate is true. The predicates are flexible. Getstats replaces several statistical variables (e.g., mean, median, half-width of the confidence interval, standard deviation, number of tests, and the slope of the linear regression) with their actual values and then passes the predicate to Perl's `eval` function. This combination of substitution and `eval` allows arbitrarily complex predicates using simple math or Perl functions. For example, the statement "$0.05 * \$mean < \$delta$" returns true if the half-width of the Student-t confidence interval is less than 5% of the mean.

Combining Results Benchmarks often have too many numbers for a user to interpret, so Auto-pilot automatically combines results in two specific cases:

- Benchmarks that consist of multiple programs (e.g., compilations)
- Multi-threaded benchmarks

Some benchmarks consist of multiple commands. For example, to compile GCC, three steps are required: `tar` extracts the distribution, `./configure` detects information about the environment, and finally `make` builds the package. Getstats unifies the results of each command into one result per benchmark execution. Alternatively, the user can use a select transformation to analyze one specific command in isolation.

When analyzing multi-threaded data, Getstats aggregates all of the threads in one test together to create a single set of results for the test. The elapsed time that is reported is the longest running thread and the system and user times are the sum of the system and user times for each thread. This allows a benchmark that is usually single-threaded to be run and analyzed in a multi-threaded manner.

3.4 Plotting Tools

One of the most useful ways to present benchmark data is through the use of graphs. In our experience, it is desirable to have automatically-generated graphs in Encapsulated Postscript (EPS) format, so the graphs are more suitable for publication. Our requirement for automatic graph generation essentially dictates that we use a command-line tool. We have found Gnuplot to be the most flexible plotting package, but its interface is cumbersome, often requiring dozens of lines of code and a specific data format to create a simple plot. Auto-pilot uses a wrapper script, *Graphit*, that generates both line and bar charts using Gnuplot. Although Gnuplot is our preference, everyone has their own favorite plotting tool. Each plotting wrapper script is specific to the plotting tool used, but Getstats can produce CSV output which can easily be imported into other programs (e.g., Excel).

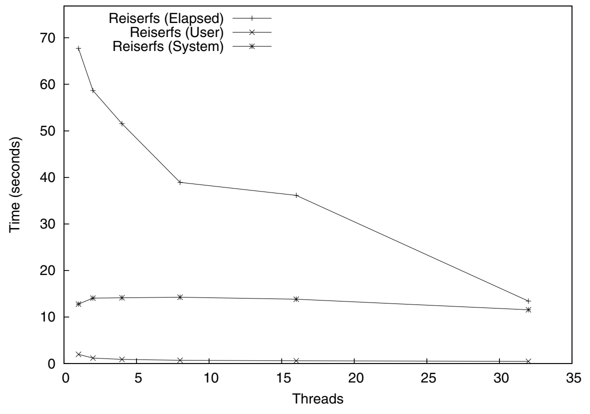


Figure 4: The results for Postmark on Reiserfs, 1–32 threads

A sample line graph can be seen in Figure 4. This graph shows the results from Postmark running on top of Reiserfs with 1–32 threads. The elapsed time decreases as threads are added, because the kernel can better schedule I/O operations. However, the user and system CPU time remained relatively constant because the same amount of work is being done.

This graph was generated with the following command line:

```
graphit --mode=line \
  --components=Elapsed,User,System \
  --graphfile=reiserfs-pm.eps \
  -k 'top right' -f 17 \
  --xlabel Threads --ylabel "Time (seconds)"  \
  Reiserfs 'reiserfs:*.res'
```

The mode argument specifies a line graph (the other supported mode is a bar graph). The `--components` parameter informs Graphit which quantities are of interest. The `--graphfile` parameter specifies an output file. The next three parameters are optional: `-k` specifies the legend location (Gnuplot refers to the legend as a key); `-f` increases the font size to 17 points; and xlabel and ylabel specify axis labels. The last two parameters define a series. In this case it is named Reiserfs, and the values are from the files named `reiserfs:*.res`. This command line replaces an 11 line Gnuplot script, with

566 characters (a savings of 70%). More importantly, *Graphit* creates a properly-formatted Gnuplot data file for the actual series.

Creating bar graphs in Gnuplot is not straightforward. Rather than defining series and values as is done in other plotting tools, you must create an artificial x-axis and locate each bar and its label along this axis. To properly space and locate these bars without automated tools is tedious. Graphit automatically constructs this artificial x-axis and spaces the bars appropriately. Using command-line arguments, the width of the bars, the gap between each bar, the gap between each group of bars, and several other parameters can be controlled easily. An example Graphit bar graph is shown in Figure 5. This graph shows the same Postmark results for Ext2, Ext3, and Reiserfs that are shown in Figures 2 and 4. Graphit automatically reads the results files, stacks user CPU time over system CPU time; determines bar widths, spacing, and error bars; formats a Gnuplot data file; generates a Gnuplot script; and finally runs Gnuplot. We ran the following command to generate the graph:

```
graphit --mode=bar \
  --ylabel 'Time (seconds)' -f 17 \
  --components=Elapsed,User,System \
  --graphfile=pm.eps Ext2 ext2:1.res \
  Ext3 ext3:1.res Reiserfs reiserfs:1.res
```

This single command replaces a 15-line 764-character Gnuplot script, and, more importantly, generates a data file with appropriate spacing along the artificial axis. Graphit has several spacing options to control the width of each bar, the gap between bars (e.g., System and User could be next to each other), and the gap between sets of bars (e.g., Ext2 and Ext3). Figure 6 is another example Graphit bar graph using the same options but with more data sets, so the bars are thinner. We also passed "`--rotate 45`" to Graphit to rotate the X-axis labels.

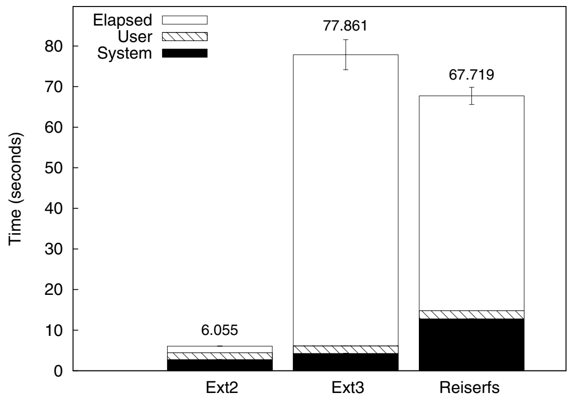


Figure 5: Postmark on Ext2, Ext3, and Reiserfs.

3.5 Checkpointing Across Reboots

Operating system code behaves very differently when data is already cached (called a warm cache) and when data is not yet in the cache (called a cold cache). This concern is particularly acute for file systems, which depend on caches to avoid disk or network operations, which are orders of magnitude slower than in-memory operations. Creating a warm cache situation is relatively easy: to create n warm cache runs, you can run the test $n+1$ times in sequence and discard the first result, which is only used to warm the cache. However, testing a cold cache situation is more difficult. For a perfectly cold cache, all OS objects must be evicted from their caches between each test. For file systems research, researchers unmount the test file system between each test. This invalidates inodes, directory name lookup caches, and the page cache for the tested file system. Unfortunately, this approach has two key disadvantages. First, even though inodes (and other objects) are invalidated, they may not be deallocated. This can expose subtle bugs when inodes are not properly cleared before reuse. The second problem is that the kernel often keeps a pool of unused objects for faster allocation, so the second run may use less system time because it does not need to get raw pages for object caches. For these two reasons, we have additionally run our version of chill, described in Section 3.3, to deallocate these objects.

To get a truly cold cache, a reboot is required because all operating state is reset after a reboot. The disadvantage of a reboot is that it takes a long time and usually requires manual intervention. We have designed Auto-pilot so that it can serialize all of its state into a plain text file using the CHECKPOINT directive. The CHECKPOINT directive is similar to UNIX fork in that it sets a variable to "0" after writing a checkpoint, but the variable is set to "1" after resuming from where the checkpoint left off. After writing a checkpoint, an Auto-pilot script can reboot the machine. After the machine's initialization process completes (at the end of `/etc/rc.d/rc.local`), we check whether the checkpoint file exists. If the checkpoint file exists, is owned by root, and is not world-writable, then we start Auto-pilot and it resumes from where benchmark execution left off.

We wanted to quantify the differences between various methods of cooling the cache. We ran a recursive `grep -q` benchmark over the GCC 3.4.3 source tree. We chose grep because it is a simple read-oriented benchmark, yet has a significant user component. We used the six following configurations:

Sequential The benchmark is run repeatedly with a warm cache.

Chill Chill is run between each iteration of the benchmark.

Remount The test file system is remounted between each iteration of the benchmark.

Remount+Chill The test file system is remounted between each iteration of the benchmark.

Reboot The machine is rebooted between each iteration of the benchmark.

Reboot+Chill The machine is rebooted and Chill is run between each iteration of the benchmark. We used this configuration because the initialization process after reboot may have caused some objects to be loaded into the cache, and chill may evict them.

All tests were run on a 1.7Ghz Pentium IV with Fedora Core 2 and a vanilla 2.4.23 kernel. The test partition was on a Western Digital 5,400 RPM IDE disk. Each test was run 10 times. The elapsed, system, and user time results are shown in Figure 6, and the error bars show the 95% confidence intervals for elapsed time.

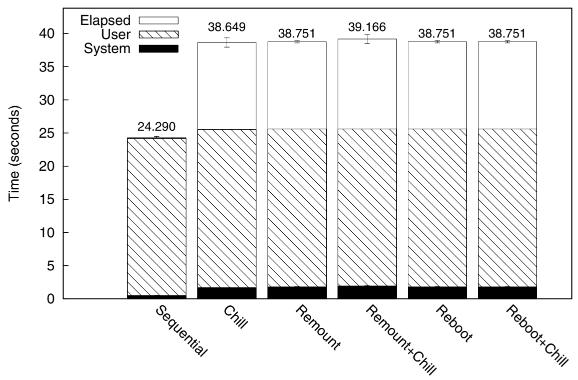


Figure 6: `grep -r` *with various cache-cooling methods.*

In all cases user time differences were indistinguishable. This was expected, because `grep`'s processing does not change with the different configurations. The Sequential configuration had the lowest elapsed and system time, because it does not need to perform any I/O to read objects from disk. For Chill and Remount, System time was not distinguishable, but Chill used 13% less system time, so remounting is a more effective way of cooling the cache. Remount+Chill is indistinguishable from remount. Reboot was 4.3% slower than remounting the file system, and system time was indistinguishable. Reboot is not distinguishable from Reboot+Chill. Therefore we conclude that rebooting the system is in fact the best method of cooling the cache.

3.6 Benchmarking Script Hooks for NFS

We have found that we continually enhance our Auto-pilots benchmarking scripts, yet each project often needs its own slightly different setup and cleanup mechanisms or needs to measure a new quantity. For example, our tracing file system needed to measure the size of the trace file that each test generated [2]. This made the scripts hard to maintain. Projects that were started earlier used older modified versions of the benchmarking scripts. As the scripts improved, these older projects would have progressively more out of date scripts. Recently, we have redesigned Auto-pilot's benchmarking scripts to be extensible and provide hooks in critical locations. Our goal was to allow developers to test new file systems and add new measurements to the benchmarking scripts, without having to modify the originals.

To do this, Auto-pilot automatically reads in all files in `$APLIB/commonsettings.d` (`$APLIB` is an environment variable that is set to the path where the scripts were installed, but additional paths can be added to it). These files define hooks for various events such as mounting a file system, unmounting a file system, beginning a measurement, and ending a measurement.

We prototyped a modified NFS client and server, and therefore did not use the standard NFS benchmark, SPEC-SFS [19], because it does not use the local client (it hand-crafts RPCs instead). To benchmark our modified client and server, we used Auto-pilot to run standard file system benchmarks, such as Postmark. We have included the hooks we used for this project as an example of what can be done with our hooks.

The simplest hook we have adds support for mounting an NFS file system. The hook requires two environment variables to be set (the server and the path to mount on the server). The hook also loads and unloads the NFS file system module (unless it is built into the kernel).

Because we were running a modified NFS server, we needed to coordinate with the server and measure the CPU time used by the NFS server (it does not make sense to measure elapsed time, because the server runs for precisely as long as the client). To coordinate between the client running the benchmarks and the server, we used SSH to run remote commands on the server. This follows the same model that SPEC SFS uses, but replaces RSH using `rhosts` with SSH using public-key based authentication. In the mount hook, we copy the module to the server and then restart it. In the unmount hook we unload the module. We also added a remote process measurement hook. Each time a client process is measured, we run a command on the NFS server to record the amount of CPU time used by nfsd. After the client process is terminated, we run another command on the server to report the difference in CPU time used.

The current scripts make it relatively easy to benchmark one client machine accessing one server, which is a relatively common benchmarking case. If you wanted to benchmark one server with multiple clients, then you would need to write new scripts in which the server runs Auto-pilot and remotely executes the benchmark on the clients. The server-side scripts would need to be written from scratch, but the client-side scripts could be very similar to the current Auto-pilot benchmarking scripts.

4 Conclusion

Auto-pilot provides a useful set of tools for accurately and informatively benchmarking software. Auto-pilot configurations are powerful scripts that describe a series of tests, including multi-threaded versions of traditional benchmarks. Auto-pilot includes sample scripts for various compile-based benchmarks and Postmark. Auto-pilot's flexible infrastructure allows many other tests and measurements to be added. For example, we include hooks for testing over NFS, measuring I/O operations and CPU time used by background processes. We have also used many different benchmarks aside from the ones included in the package (e.g., `grep` and other custom benchmarks).

We present results in an informative manner that can easily be used with Gnuplot. Getstats generates easy-to-read tabular reports, automatically displays outlying points, detects memory leaks, and runs statistical tests. In combination with Auto-pilot, Getstats can evaluate predicates to run tests until an arbitrarily complex condition is satisfied, thereby saving time on the testbed. Graphit processes results files and automatically creates Gnuplot scripts with properly formatted data files to create bar and line graphs.

Auto-pilot can checkpoint its state, reboot the machine, and resume running tests from where it left off. Rebooting provides a colder cache than simply remounting the file system, and requires no manual intervention on the part of the benchmarker. When benchmarks fail, Auto-pilot can automatically send email or text pages to the benchmarker to prevent the testbed from lying idle.

We have used Auto-pilot for over three years and for eighteen projects. Auto-pilot saved us many days of work in collecting performance results and reduced debugging cycles by exposing bugs more quickly.

Auto-pilot is released under the GPL and can be downloaded from `ftp.fsl.cs.sunysb.edu/pub/auto-pilot/`. Auto-pilot contains 5,799 lines of code. The Auto-pilot Perl script has 936 lines; Getstats has 1,647 lines; Graphit has 587 lines; the C utilities have 758 lines; the configuration files have 221 lines; and our shell scripts have 1,650 lines. All of our Perl scripts also have basic Perldoc formatted documentation that provides information on simple usage. We also include a full user manual that describes more detailed usage and has some brief tutorials in PDF and GNU `info` formats.

4.1 Future Work

We plan to add support for regular system snapshots, which can then be correlated with the output of the tests. This will allow a person conducting benchmarks to determine the context that the test was executed in. These snapshots will include kernel messages, memory usage, CPU usage, and other vital statistics.

Getstats currently uses a two-sample t-test to compare different results files. A t-test makes three assumptions:

- The samples are independent
- The samples are normally distributed
- The samples have equal variance

Auto-pilot scripts attempt to ensure that each sample is independent of the other samples by purging cached data through remounting and rebooting. The last two assumptions must currently be verified by the benchmarker. If the sample size is sufficiently large (roughly greater than 30 samples), the t-test will still be appropriate. The third assumption must still be verified. We plan to automatically run an F-test on the variances. If the test concludes that the variances are not equal, then Getstats will use an approximate t-test for samples with unequal variances instead of the standard t-test.

Acknowledgments

Joseph Spadavecchia developed the first simple version of our benchmarking Perl script that has since morphed into the current Auto-pilot. Amit Purohit, Kiran-Kumar Muniswamy-Reddy, Michael Martino, Avishay Traeger, and others in our group have provided valuable feedback when using the system to benchmark their projects.

This work was partially made possible by NSF CAREER award EIA-0133589, NSF Trusted Computing Award CCR-0310493, and HP/Intel gift numbers 87128 and 88415.1.

References

[1] AIM Technology. AIM Multiuser Benchmark - Suite VII Version 1.1. `http://sourceforge.net/projects/aimbench`, 2001.

[2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.

[3] Bell Laboratories. *prof*, January 1979. Unix Programmer's Manual, Section 1.

[4] M. Blaze. A cryptographic file system for Unix. In *Proceedings of the fir st ACM Conference on Computer and Communications Security*, 1993.

[5] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case

Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224. ACM Press, June 1997.

[6] R. Bryant, R. Forester, and J. Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 259–274, Monterey, CA, June 2002.

[7] S. Gaede. Perspectives on the SPEC SDET benchmark. `www.specbench.org/sdm91/sdet/`, January 1999.

[8] The GCC team. *GCC online documentation*, 3.3.2 edition, August 2003. `http://gcc.gnu.org/onlinedocs/`.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, June 1982.

[10] A. Ho and D. Coppit. *Benchmark::Timer - Benchmarking with statistical confidence* , December 2004. User Contributed Perl Documentation, Section 3.

[11] IBM. Software testing automation framework STAF. `staf.sourceforge.net`, 2001.

[12] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[13] Open Source Development Labs. Scalable test platform. `www.osdl.org/lab_activities/kernel_testing/stp/`, 2004.

[14] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 279–295, January 1996.

[15] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[16] D. Phillips. A directory index for EXT2. In *Proceedings of the 5th Annual Linux Showcase & Conference*, pages 173–182, November 2001.

[17] The OpenSSL Project. Openssl: The open source toolkit for SSL/TLS. `www.openssl.org`, April 2003.

[18] H. V. Riedel. The GNU/Linux CryptoAPI site. `www.kerneli.org`, August 2003.

[19] SPEC: Standard Performance Evaluation Corporation. SPEC SFS97_R1 V3.0. `www.spec.org/sfs97r1`, September 2001.

[20] Sun Microsystems, Inc. *Chill – remove useful pages from the virtual memory cache*. SunOS 4 Reference Manual, Section 8.

[21] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.

Interactive Performance Measurement with *VNCplay*

Nickolai Zeldovich Ramesh Chandra
Computer Science Department
Stanford University

{nickolai, rameshch}@cs.stanford.edu

Abstract

Today many system benchmarks use throughput as a measure of performance. While throughput is appropriate for benchmarking server environments, response time is a better metric for evaluating desktop performance. Currently, there is a lack of good tools to measure interactive performance; although several commercial GUI testing tools exist, they are not designed for performance measurement.

This paper presents *VNCplay*, a cross-platform tool for measuring interactive performance of GUI-based systems. *VNCplay* records a user's interactive session with a system and replays it multiple times under different system configurations; interactive response time is evaluated by comparing the times at which similar screen updates occur in each of the replayed sessions. Using *VNCplay* we studied the effect of processor speed and disk load on interactive performance of Microsoft Windows and Linux. These experiments show that the same user session can have widely varying interactive response times in different environments while maintaining the same total running time, illustrating that response time is a better measure of interactive performance than throughput. The experimental results make a case for a response time measurement tool like *VNCplay*.

1 Introduction

Most performance evaluation studies today use throughput benchmarks to quantify system performance. However, throughput is not an appropriate performance metric for user desktops. User interface studies [12] have shown that response time, rather than throughput, is the right measure of interactive performance. We believe that the lack of research studies on interactive performance is due to the lack of tools that measure response time.

In designing *VNCplay*, we came up with the following characteristics that a good tool for benchmarking interactive performance should have. First, it should be able to record an interactive session and replay it multiple times under different environments, so that the same repeatable workload can be used to meaningfully compare interactive performance. Second, the tool should be able to extract a measurement of response time from replayed sessions to quantify the observed performance. Finally, it is desirable that the tool not be tied to a specific platform or GUI toolkit, so that researchers can compare the performance of various systems.

We developed a tool, called *VNCplay*, that satisfies the above criteria. This tool was developed out of a need to evaluate the interactive performance of the Collective system [10, 2]. There are several commercial GUI testing tools that provide some amount of recording and replay capability [15, 17, 14]. However, they are mainly intended for testing and do not replay reliably when the system is slow. Furthermore, they only support specific toolkits and do not have a facility for extracting measurements from the replays.

VNCplay uses the VNC remote display protocol [9] for recording and replaying sessions. Since VNC servers are available for many platforms, *VNCplay* supports a wide variety of systems. In addition, we developed an analysis technique to extract useful response time measurements from the replay sessions. We used *VNCplay* to evaluate the effect of processor speed and disk I/O on response times of interactive applications. This evaluation shows that throughput benchmarks are not sufficient in measuring interactivity and validates the need for a tool such as *VNCplay*.

The rest of the paper describes *VNCplay* in more detail. A short user's view of *VNCplay* is presented in Section 2. Section 3 explains the design and implementation of *VNCplay*. In Section 4 we demonstrate the use of *VNCplay* in evaluating the interactive performance of Microsoft PowerPoint and OpenOffice Impress over a range of CPU speeds and I/O loads, as well as evaluating the interactive performance of different Linux disk I/O schedulers. Section 5 describes some ideas for future work.

Related work is discussed in Section 6 and we conclude in Section 7.

2 User's View of *VNCplay*

Using *VNCplay* to record and replay interactive sessions is as easy as using a VNC client. To record a session, a user runs

```
$ vncplay record server:port trace.vnc
```

This command brings up a VNC client connected to the specified server. The user performs the workload to be recorded, and closes the VNC client at the end of the workload. In case of the command shown, the user's workload is saved into the file *trace.vnc*. To replay a recorded workload, the user runs

```
$ vncplay play server:port trace.vnc out.rfb
```

The workload from *trace.vnc* will be replayed against the specified VNC server, and the entire session output will be saved into a log file *out.rfb* for later review or analysis. A view-only VNC client (one that will not accept keyboard and mouse input from the user) will be displayed while the replay is taking place, to provide the user with visual feedback. To allow for unattended session replay, the password for the VNC server can be saved to a file and passed to *vncplay* by specifying the *–pwfile* option on the command line.

To obtain interactive performance metrics from replayed sessions, the user first runs an analysis stage on the session log files, which takes some time to run (on our computer it takes approximately as long as replaying each of the sessions):

```
$ vncplay analyze out1.rfb ...  > analyze.out
```

The command shown above produces intermediate analysis results in *analyze.out*, which can be then used to generate graphs like the ones presented later in this paper. For example, a cumulative distribution function of response times can be generated by

```
$ vncanalyze cdf analyze.out > cdf.out
```

The resulting *cdf.out* can be plotted by a tool like *gnuplot* [5].

3 Design and Implementation

VNCplay consists of three components: a recorder, a replayer, and an analyzer. The recorder and replayer provide reliable replay of interactive sessions. The analyzer takes replayed sessions and extracts interactive performance metrics from them. All of the components are based on a modified version of the TightVNC [13] Java client. The following sections describe some details of the design and implementation of *VNCplay*.

3.1 Recording and Playback

Based on our past experience with interactive session replay tools, such as [15, 7, 19, 1], we observed that one of the biggest problems that all of the tools have in common is correctly replaying mouse clicks. Most tools replay each input event, such as a mouse click, at exactly the same time that it occurred during recording. When replaying a session on a slower system, such an approach is not appropriate, as it can easily lead to mouse clicks being delivered to the wrong application. For example, suppose that during recording, the user brings up a window and clicks on a button in that window. During replay on a slower system, the window might take a few more seconds to appear. A purely time-based replay tool would click on the background image, where the button should have been, without waiting for the window and button to appear, resulting in different behavior in the replay than in the recorded session. This is not acceptable; therefore *VNCplay*'s recorder and replayer focus largely on reliable delivery of mouse events. Section 5 discusses reliable replay of sessions with both keyboard and mouse input.

The VNC recorder acts like a normal VNC client, except that it records the user's activity into a VNC replay log, such as the one shown in Figure 1. The log contains input events (mouse clicks and movements, and key presses) along with timestamps of when these events occurred. For each mouse click event, a snapshot of the screen is taken by the recorder and written to the replay log. These screen snapshots will enable the VNC replayer to reliably deliver the associated mouse click events.

During playback, the replayer connects to a VNC server as a client, and reads events from the VNC replay log. Events from the replay log are processed according to their timestamps; input events such as key presses and mouse movements are sent to the VNC server directly. For mouse clicks, the VNC replayer waits for the screen image to match the corresponding screen snapshot in the replay log before sending the mouse click event. The screen snapshot captures the visual state of the system when the user clicked the mouse during recording. By waiting for the screen image to match the snapshot, we ensure that the system has reached the same state during replay as during recording, at which time the mouse click can be safely delivered.

The replayer records all of the screen updates received from the server during replay into a log file [8] for later

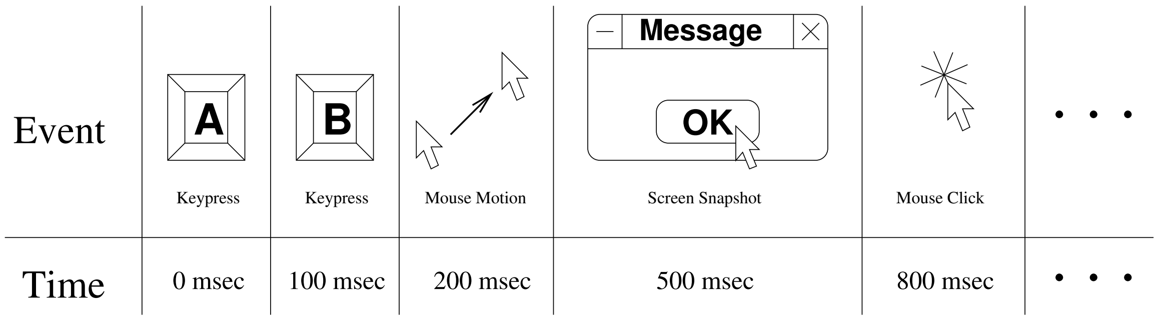


Figure 1: Example of a replay log used to record and play back VNC sessions. Keys ”A” and ”B” were pressed at 0 and 100 msec respectively, the mouse cursor was moved at 200 msec, a screen snapshot was taken at 500 msec, and a mouse click happened at 800 msec.

analysis or debugging. This allows the analysis of the replayed session to be performed at a later time offline, as it can be computationally intensive.

3.2 Reliable Playback

To reliably replay the same session multiple times, the system must behave repeatably – that is, at each replay, it should start from the same state, and given the same input, it should provide the same output. In working with *VNCplay*, we have found that complex desktop environments, such as Windows or Linux, are not fully deterministic from the point of view of VNC. This section describes some of the problems we encountered in achieving reliable session playback, and workarounds we have implemented to address them.

We found that a snapshot of the entire screen is often difficult to match during replay; non-deterministic elements like the system clock or tooltips are usually different between recording and replay, and get in the way of perfect reproducibility. *VNCplay* uses screen snapshots to ensure that mouse clicks are delivered to the same UI element during replay as during recording. For this purpose, we have found that it suffices to take a snapshot of just the screen area around the mouse cursor. For example, if during recording the user clicks on an “OK” button, we only need to take a snapshot of the button to ensure that we click on it correctly during replay. In our current implementation, we use a square area of about 10 pixels by 10 pixels around the cursor for screen snapshots. This significantly improves *VNCplay*’s ability to reliably replay sessions, by avoiding non-deterministic tooltips and other changing screen elements.

VNCplay attempts to deliver mouse click events reliably during playback, but other events, such as mouse motion, are simply replayed at the pace at which they were recorded. This resulted in some surprising behavior in a situation where mouse motion does matter. Some GUI elements, such as menus, change appearance when the mouse is located over them – for example, the menu item that the mouse is pointing to might be highlighted. When a menu is slow to open, *VNCplay* will move the mouse cursor to where it expects the menu item to appear, before the menu item actually appears on the screen. As it turns out, in Windows, menu items “notice” that the mouse is pointing to them and highlight themselves only when the mouse moves. Thus, if the mouse is already pointing to a menu item by the time the menu item is drawn, the menu item will fail to notice that it should be highlighted. In turn, *VNCplay* will be unable to match the screen snapshot taken during recording, which shows a highlighted menu item, and replay will stall. To fix this problem, the replayer wiggles the mouse cursor by one pixel while waiting for a screen snapshot to match. This triggers “on-mouse-motion” callbacks in such GUI elements, allowing *VNCplay* to proceed with playback.

VNC is a very simple remote frame buffer protocol, in which the server provides periodic screen updates to the client at arbitrary intervals; it is up to the server to decide when a screen update should be sent to the client. The VNC protocol provides a kind of “eventual consistency” guarantee: when an image appears on the server’s screen and stays there, a screen update containing that image will eventually be sent to the client. In particular, consider the effect of this remote frame buffer model on the updates seen by a VNC client during the rendering of a complex user interface. A VNC client might receive a screen update for every intermediate step of rendering the screen image, such as the steps shown in Figure 2, or it might only see one screen update representing the final state – such as step 3 in the figure.

This behavior of VNC screen updates complicates the process of taking a screen snapshot in *VNCplay*, because a screen snapshot is something that we expect to see each and every time we replay the recorded session. If the recorder takes a snapshot using an intermediate screen update, during replay the VNC server might not send us

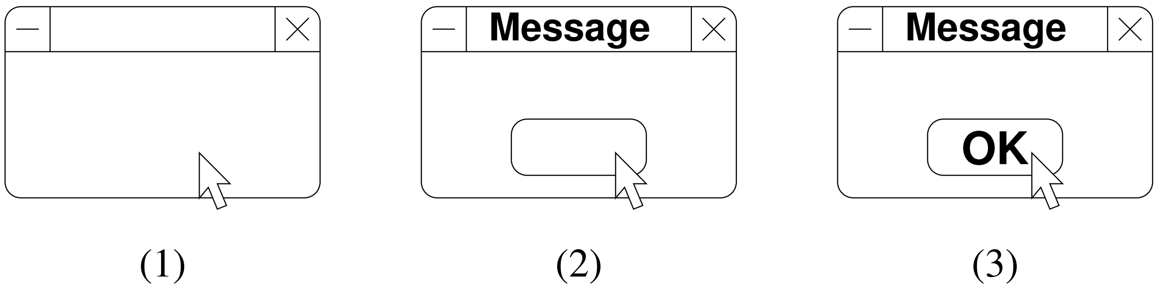


Figure 2: Three consecutive screen updates, as a dialog box is being rendered, that could be combined by the VNC server. VNC quiescing ensures that only the last screen update in such a sequence is used for a screen snapshot.

the same intermediate screen update, instead choosing to send only the final state and thereby preventing the re-player from matching the screen snapshot. For example, if we take a screen snapshot using step 1 or 2 in Figure 2 during recording, the server may choose to only send us the third screen update from the figure, preventing the re-player from successfully matching the screen snapshot. This means that if replay is to be reliable, screen snap-shots taken by the VNC recorder should not be intermediate screen states.

To get around the problem of intermediate screen states, we implement VNC *protocol quiescing* during recording. When the recorder decides to take a screen snapshot, it temporarily blocks input events from being sent to the VNC server, and waits a short period of time for the system to process all prior user input and come to some final state; in other words, quiesce. After the screen has quiesced, the recorder takes a screen snapshot, and any blocked input events are sent to the VNC server. This technique produces screen snapshots which can be reliably observed during playback, and allows for robust re-play of sessions. We have measured the time required for the operating system and the VNC server to quiesce under workloads such as a user using Microsoft PowerPoint or Word. On a 100 Mbps local area network, the VNC screen image quiesces within 100 milliseconds; *VNCplay* conservatively waits for 150 milliseconds before taking a screen snapshot.

3.3 Performance Analysis

The main metric that we wish to obtain from the interactive replay experiments is the response time for each input event. We implemented an analyzer that compares a set of replayed sessions and extracts interactive response times for various input events.

The analyzer looks for similar screen updates between the replayed sessions; for example, if the user opens a menu in the recorded session, the analyzer would find the times at which the menu opened in the different replays. For each matching screen update, it finds the nearest pre-ceeding input event in all of the sessions, and assumes that this input event caused the screen update. The time difference between the screen update and the input event in each session is taken as the interactive response time for that input event in that session.

To make the analyzer run in acceptable space and time, we had to make a few optimizations. First, the resolution of all screen updates is scaled down (currently by a factor of four in each dimension). This reduces the size of each screen update by a factor of 16, without impacting the accuracy of screen matching – the features we want to match are larger than 4 pixels. Next, screen updates that happen at the same time are coalesced, reducing the number of screen updates that need to be scanned. Lastly, only significant screen updates are analyzed to find corresponding matches – currently the threshold we use is at least 2% pixel difference from the previous analyzed update. This optimization prevents the analyzer from analyzing periods of little or no activity (for example, only mouse movement). As a result of these optimizations, the analyzer can compare two typical interactive sessions on a 2.4GHz Pentium IV computer with 1GB of memory in about the same time it takes to replay the sessions.

4 Evaluation

This section describes our experience using *VNCplay* to evaluate interactive performance of Microsoft Windows and Linux. We subject these systems to various workloads and compare the interactive performance under these scenarios.

We performed four sets of experiments. The first two experiments measure effect of processor speed and disk I/O on interactive performance. The third experiment demonstrates that *VNCplay* can reliably replay interactive sessions over a wide range of system response times by running workloads on a system with an extremely slow disk, which increases the total runtime by a factor of 10. Finally, the fourth experiment shows that we can measure interactive performance of a conventional Linux machine without the use of VMware, by evaluating the interactive performance effects of different disk I/O schedulers in Linux 2.6.

For the first three experiments, we used VMware's GSX Server [16] to run the system to be measured. VMware simplifies the task of making an identically-configured system for each experimental run. It also provides an efficient and platform-independent VNC server that is connected to the virtual machine's console. These experiments were done on a Thinkpad T42p laptop with a 2.0 GHz Pentium M processor, 1GB of memory and a 7200rpm 60GB hard drive.

For experiments with VMware virtual machines, we used the VNC recorder to record a user session in a virtual machine containing Microsoft Windows XP and Microsoft Office 2003, and another virtual machine containing Fedora Core 1 Linux and OpenOffice. Both sessions lasted about 6 minutes and consisted of a user creating a presentation, either in PowerPoint or OpenOffice Impress. These sessions was replayed in a variety of environments and the resulting session logs analyzed for interactive performance.

4.1 Effect of Processor Speed

We used the Enhanced Speedstep capabilities of the Pentium M processor to vary the processor speed from 300 MHz to 2.0 GHz and replayed both the PowerPoint and OpenOffice sessions in each of these scenarios. A point to note here is that the Pentium M processor running at a reduced speed does not accurately simulate the performance of an older processor that normally runs at that speed. This is because other characteristics of the processor, such as cache size and memory bus speed, remain unchanged.

Figures 3 and 4 show the total running time of the PowerPoint and OpenOffice sessions for various processor speeds. The running time stays more or less the same, with the difference between fastest and slowest times being less than 1%.

However, the interactive response times of individual events are very different for different processor speeds. Figures 5 and 6 show a CDF plot of the response times for sessions replayed under two different configurations: a simulated 300 MHz system and a 2.0 GHz system. The CDF plot shows the fraction of time that the system's response to user input was within a given value. For instance, a point with an x-axis value of 1 second and y-axis value of 90% would indicate that 90% of the time, the interactive response time was within 1 second.

These figures clearly show that both PowerPoint and OpenOffice running at 300 MHz respond much slower than running at 2.0 GHz, and a user would find it to be significantly sluggish. For example, in the Linux OpenOffice environment the response time at the 40th percentile when running at 300 MHz is about five times the response time as when running at 2.0 GHz. The slowdown is further illustrated by Figures 7 and 8. These figures show the ratio of response times of events in OpenOffice and PowerPoint running at 300 MHz, to the response times of the same events while running at 2.0 GHz. The OpenOffice plot shows that most of the events are slowed down by a factor of two to factor of five, while for PowerPoint the slowdown is much more modest. This shows that OpenOffice running under Linux requires more CPU resources than PowerPoint under Windows XP.

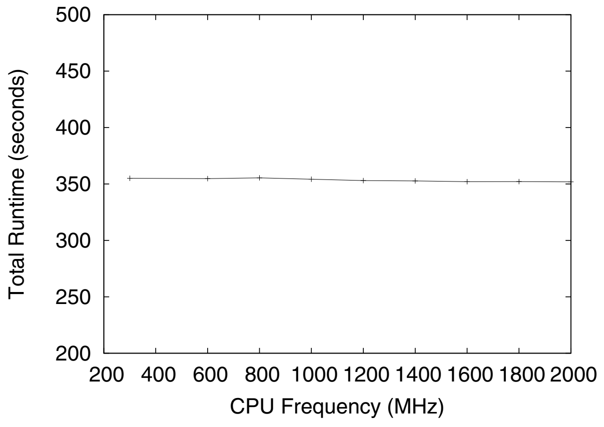


Figure 3: Total running time of a Microsoft PowerPoint session at various processor speeds

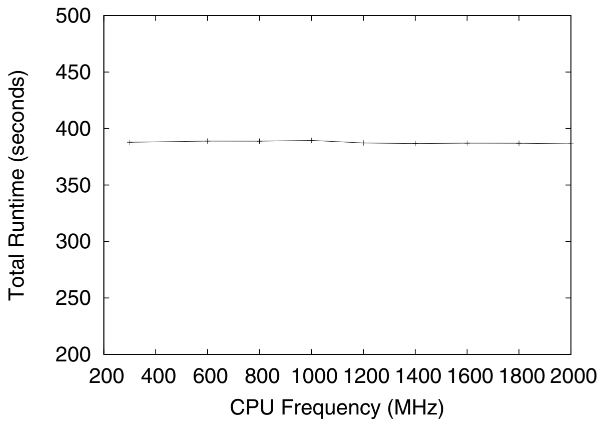


Figure 4: Total running time of a Linux OpenOffice session at various processor speeds

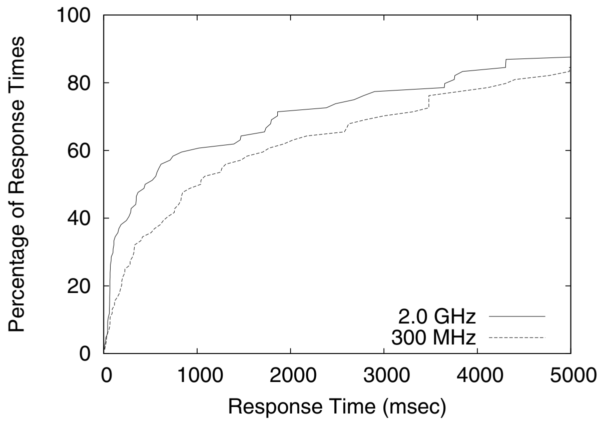


Figure 5: CDF plot of interactive response times for Microsoft PowerPoint under different conditions: on a 2.0 GHz machine and on a simulated 300 MHz machine. Each line shows the fraction of interactive response times (vertical axis) that are within a certain value (horizontal axis).

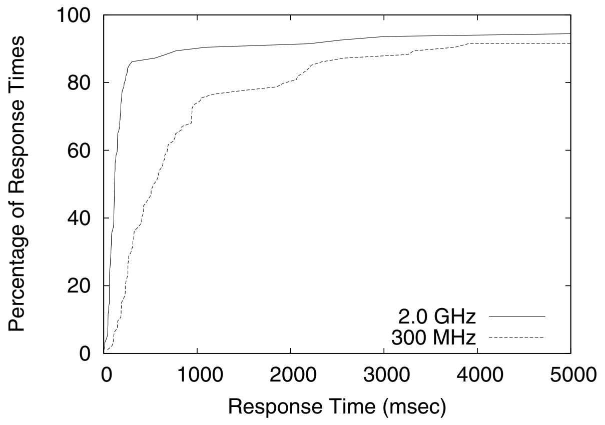


Figure 6: CDF plot of interactive response times for Linux OpenOffice under different conditions: on a 2.0 GHz machine and on a simulated 300 MHz machine. Each line shows the fraction of interactive response times (vertical axis) that are within a certain value (horizontal axis).

Note that inspite of the significant differences in response times between the 300 MHz and the 2.0 GHz runs, the total run time is the same in both OpenOffice and PowerPoint. This is because the interactive sessions consist of considerable amount of idle time between events (i.e., user think time). The extra latency in the 300 MHz sessions is absorbed by this idle time, leaving the total running time unchanged. This clearly demonstrates that total run time (and hence throughput benchmarks) are not appropriate for studying interactive responsiveness of systems and makes a case for tools like *VNCplay*.

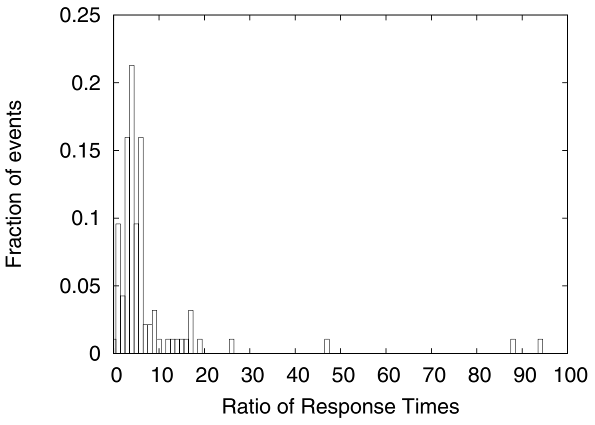


Figure 7: Histogram of the ratio of interactive response times for Linux OpenOffice on a simulated 300 MHz machine and a 2.0 GHz machine.

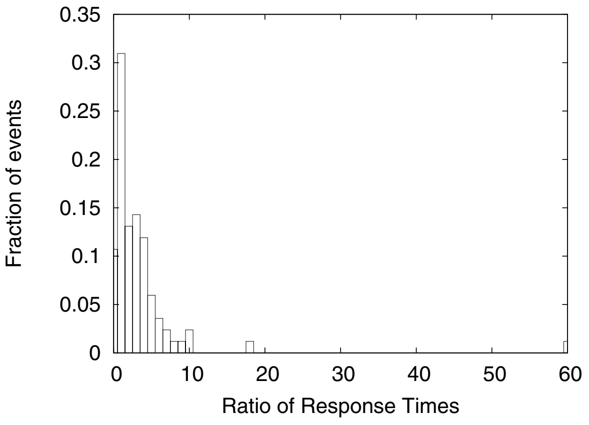


Figure 8: Histogram of the ratio of interactive response times for Microsoft PowerPoint on a simulated 300 MHz machine and a 2.0 GHz machine.

To compare across a range of processor speeeds, we plot the 25th, 50th, and 75th percentile response time latencies at various processor speeds in Figures 9 and 10. From these figures, we see that there is a minimum processor speed (between 300 MHz and 600 MHz) below which the interactive performance of office workloads de-

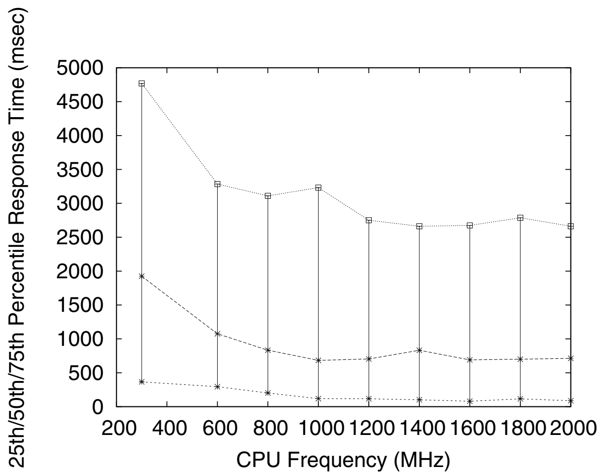


Figure 9: 25th/50th/75th percentiles of the interactive event latencies for Microsoft PowerPoint at various processor speeds

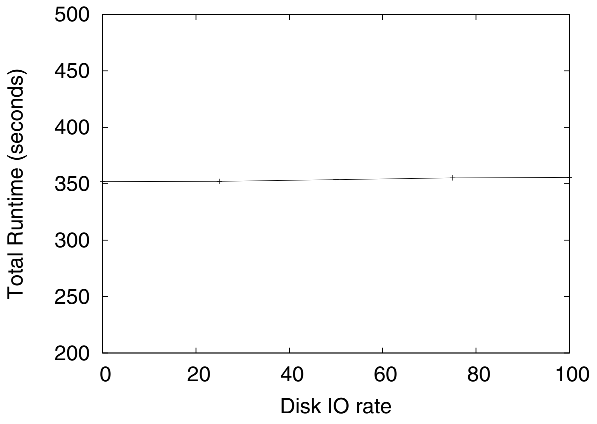


Figure 11: Total running time of the PowerPoint session at various disk I/O rates

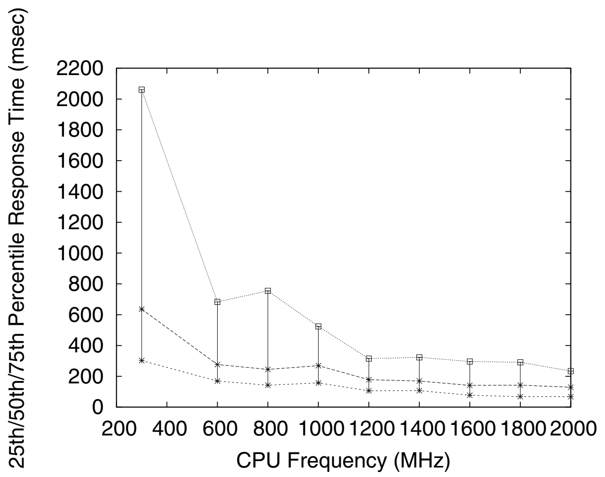


Figure 10: 25th/50th/75th percentiles of the interactive event latencies for Linux OpenOffice at various processor speeds

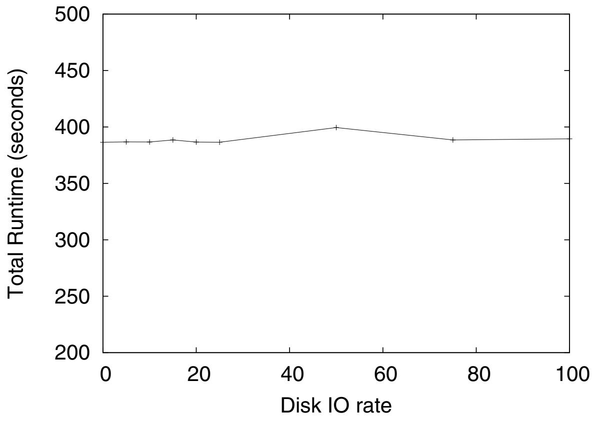


Figure 12: Total running time of the OpenOffice session at various disk I/O rates

grades rapidly. Increases in processor speed beyond this point provide a gradually diminishing increase in returns, as is to be expected.

4.2 Effect of Disk I/O

We measure the effect of disk I/O on the interactive performance of Microsoft Windows and Linux systems by injecting background disk activity and replaying the PowerPoint and OpenOffice sessions. We wrote a small utility that performs background disk I/O at a specified rate. Each disk I/O is a 32 KB read from the disk.

Figures 11 and 12 show the total run times for various disk I/O rates. As in the previous experiment, the differences in total run times are very small. In Figures 13 and 14 we can see the distribution of interactive response times in two sessions: one with no extra background disk activity and the other with a rate of 100 disk I/Os per second. The response times for the latter session are higher than the ones in the former session. Although the interactive response times of the system are noticeably different in the two cases, the total runtime remains the same, underscoring the need for an interactive performance measurement tool like *VNCplay*.

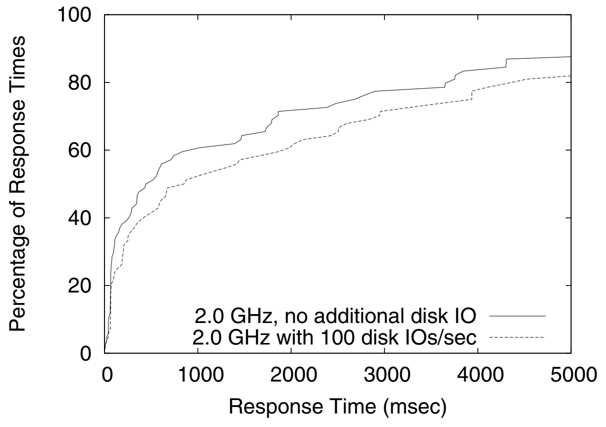


Figure 13: CDF plot of interactive response times for Microsoft PowerPoint under different conditions: on a 2.0 GHz machine with no additional disk IO and on the same machine experiencing 100 additional disk IOs per second. Each line shows the fraction of interactive response times (vertical axis) that are within a certain value (horizontal axis).

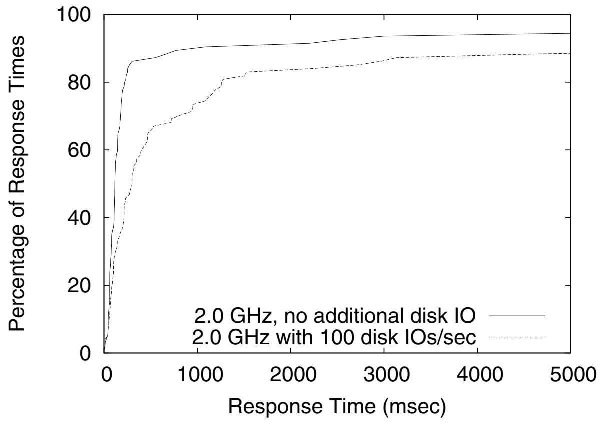


Figure 14: CDF plot of interactive response times for Linux OpenOffice under different conditions: on a 2.0 GHz machine with no additional disk IO and on the same machine experiencing 100 additional disk IOs per second. Each line shows the fraction of interactive response times (vertical axis) that are within a certain value (horizontal axis).

4.3 Reliable Session Playback

To verify that our tools can replay interactive sessions in extreme environments, we simulated an environment with an extremely slow disk subsystem. In particular, we moved the virtual disks of our experimental virtual machine onto an NFS file server connected by a simulated 1.5 Mbps down / 384 Kbps up DSL network link with 40ms round-trip latency. We then replayed a PowerPoint session and a similar Word session on this virtual machine. Although the total running time of the sessions increased from 6 minutes to approximately an hour, the sessions nonetheless completed successfully. This suggests that our replay mechanism is robust against large variations in system response time.

4.4 Replay without VMware

In the above experiments, we used VMware to run the system to be measured. In some experiments this might not be appropriate and the system might need to run on physical hardware. In this section, we demonstrate the use of *VNCplay* in one such scenario: measurement of the effect of different linux disk I/O schedulers on interactive performance. This experiment needs to be performed on physical hardware since VMware's high overhead for disk I/O can bias the experimental results.

The test machine in this case was a 2.2GHz Xeon machine with 4GB of memory, running Fedora Core 3. The system was configured to run a standard graphical login session on a VNC server for the purpose of this experiment. For each replay session, a test user account was created from scratch and the machine was rebooted to clear the buffer cache. This ensures that the test machine is brought back to the same state at the beginning of each experimental run.

For this experiment, we recorded a user session lasting about 8 minutes. This consisted of a user creating a simple presentation in OpenOffice Impress, developing a small program in the KDevelop integrated development environment, and changing his desktop background. The Linux system was configured with a different I/O scheduler on each experiment run, and we injected heavy background disk activity. The session was replayed to analyze interactive performance. We tested the anticipatory, deadline, cfq, and noop schedulers that are present in the Linux 2.6 kernel.

Scheduler	Total Runtime (min)
anticipatory	14.1
cfq	7.7
deadline	7.5
noop	7.4

Figure 15: Total running time of the user session with various I/O schedulers

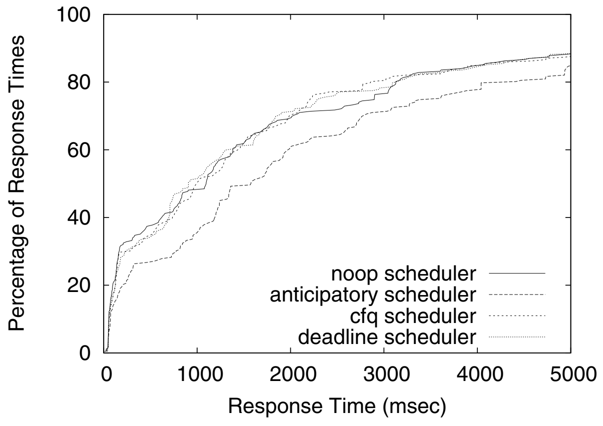


Figure 16: CDF plot of interactive response times for a desktop Linux workload using different disk I/O schedulers and a heavy background disk I/O load.

Figure 15 shows the total runtime, and Figure 16 shows the CDF plot of response times for sessions replayed under the various schedulers. As these figures illustrate, the anticipatory disk scheduler has the worst interactive performance under heavy disk load. This can be easily explained by the anticipatory scheduling algorithm. After having serviced a disk request from a process, the anticipatory scheduler waits for a short period of time for subsequent requests from the same process; any disk requests so received are given high priority. Thus this algorithm favors disk requests from background process with heavy disk activity, and hurts the performance of interactive processes.

From the results of our evaluation, we observe that latency measurements rather than total runtime are an appropriate metric of interactive performance. We also see that *VNCplay* can robustly measure interactive latency over a wide range of scenarios, including various platforms such as Microsoft Windows and Linux, and wide variations in system response time.

5 Future Work

There are a few areas in which our current recorder and replayer fall short. Our current implementation does not attempt to properly synchronize keyboard input events. To make session replay robust across a wide range of workloads, we have worked around the problem by explicitly inserting screen snapshots before keyboard input. This is done by making additional mouse clicks in the application before any keyboard input is sent.

Time-sensitive UI elements are another shortcoming of our system we hope to address in the future. For instance, Windows makes use of sub-menus that automatically expand when the mouse cursor hovers over a menu item for some period of time. The VNC replayer does not wait for the sub-menu item to appear before proceeding with the replay, and thus can sometimes go astray when replaying in a slow environment. In the current system, we explicitly click on each menu item, even if it has already expanded automatically, to ensure that session can be replayed reliably.

We are exploring the following idea to learn dependencies between input events (both keyboard and mouse) and output events from many replay sessions. After recording an interactive session on a baseline system, we replay the same workload on the baseline system with slight timing variations in the input. Using the runs that were successful, *VNCplay* learns which output events must always happen before an input event is sent, and which ones are irrelevant for the purpose of dependencies. We believe that this information can be used to make replay very robust.

An alternative approach to handling keyboard input is to use a mouse-driven keyboard input tool like the Character Map in Windows.

6 Related Work

Industry benchmarks such as Winbench and Winstone [18] measure the time to complete a fixed workload, but do not indicate how responsive the system is to user input.

Several tools available today for replaying interactive workloads are toolkit-specific [15, 17, 19, 14, 1]. In contrast, *VNCplay* is toolkit- and platform-agnostic. Most of the above tools are intended for GUI testing and require the user to manually insert delay statements for correct replay on slower systems. In addition, to the best of our knowledge, none of them provide response time measurements for the replayed sessions. *VNCplay* provides such measurements and can also be used to perform GUI testing and task automation similar to the tools mentioned above.

Interest in the research community on quantifying interactive performance has been relatively recent. Endo et. al. [4, 3] make a case for using latency as a measure for interactive performance. Recent work on measuring thin client systems [11, 6] uses response time as a measure of performance of thin client systems. However, there is no general toolkit for measuring interactive response times, and we developed *VNCplay* to fill this need.

7 Conclusions

This paper presents a tool for reliably replaying interactive user sessions by correlating screen updates with user input while recording and obeying this ordering during replay. The interactive performance of the system in each of the replayed sessions is analyzed by comparing the timestamps of matching screen updates across sessions. Our evaluation of the tool suggests that it is useful for measuring interactive performance of real systems.

8 Availability

The toolkit described in this paper is available for download at:

`http://suif.stanford.edu/vncplay/`

9 Acknowledgements

This material is based upon work supported in part by the National Science Foundation under Grant No. 0121481. We thank Constantine Sapuntzakis and Monica Lam for their input during the design and implementation of the *VNCplay* tool.

References

[1] AutoIt version 3 home page. `http://www.autoitscript.com/autoit3/`.

[2] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. The Collective: A cache-based system management architecture. In *Proceedings of 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (to appear).

[3] ENDO, Y., AND SELTZER, M. Improving interactive performance using TIPME. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (2000), pp. 240–251.

[4] ENDO, Y., WANG, Z., CHEN, B., AND SELTZER, M. Using latency to evaluate interactive system performance. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation* (October 1996).

[5] gnuplot homepage. `http://www.gnuplot.info/`.

[6] NIEH, S. J. Y. J., AND NOVIK, N. Measuring thin-client performance using slow-motion benchmarking. In *Proceedings of the 2001 USENIX Annual Technical Conference* (June 2001).

[7] Scriptable VNC session control. `http://cyberelk.net/tim/rfbplaymacro/`.

[8] RFB proxy. `http://cyberelk.net/tim/rfbproxy/`.

[9] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing 2*, 1 (January/February 1998), 33–38.

[10] SAPUNTZAKIS, C., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (December 2002).

[11] SCHMIDT, B. K., LAM, M. S., AND NORTHCUTT, J. D. The interactive performance of SLIM: a stateless, thin-client architecture. In *Proceedings of the 17th ACM Symposium on Operating System Principles* (December 1999).

[12] SCHNEIDERMAN, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, third ed. Addison Wesley Longman, 1998.

[13] TightVNC web page. `http://www.tightvnc.com/`.

[14] Sun workshop visual replay. `http://www.atnf.csiro.au/computing/software/sol2docs/manuals/visual/user_guide/Replay.html`.

[15] Rational VisualTest. `http://www.ibm.com/software/awdtools/tester/robot/`.

[16] VMware GSX server. `http://www.vmware.com/products/server/gsx_features.html`.

[17] Mercury WinRunner. `http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/`.

[18] Business Winstone. `http://www.veritest.com/benchmarks/bwinstone/`.

[19] Xnee home page. `http://www.gnu.org/software/xnee/www`.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

SAGE

SAGE is a Special Interest Group (SIG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ AMD ❖ Asian Development Bank ❖
❖ Atos Origin BV ❖ Cambridge Computer Services, Inc. ❖ Delmar Learning ❖
❖ DoCoMo Communications Laboratories USA, Inc. ❖ Electronic Frontier Foundation ❖
❖ Hewlett-Packard ❖ IBM ❖ Intel ❖ Interhack ❖ MacConnection ❖
❖ The Measurement Factory ❖ Microsoft Research ❖ Oracle ❖ OSDL ❖ Perfect Order ❖
❖ Portlock Software ❖ Raytheon ❖ Sun Microsystems, Inc. ❖ Taos ❖ Tellme Networks ❖
❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

SAGE Supporting Members

❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ Asian Development Bank ❖
❖ Fotosearch ❖ Microsoft Research ❖ MSB Associates ❖ Raytheon ❖
❖ Ripe NCC ❖ Taos ❖ Tellme Networks ❖

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-28-5